REMARKS/ARGUMENTS

By this paper, Applicant responds to the Office Action of January 13, 2005 and respectfully requests reconsideration of the application. The shortened statutory period runs through April 13, 2005. Accordingly, this response is timely.

I. Real Party in Interest

The real party in interest is ATI International SRL of Barbados, the assignee of this application. ATI International is related to ATI Technologies, Inc. of Ontario, Canada.

II. Related Appeals and Interferences

Applicant is unaware of any related appeals or interferences.

III. Status of Claims

Claims 1-83 are now pending, a total of 83 claims. Claims 1, 5, 33, 56 and 79 are independent. All rejections are traversed.

A complete copy of the claims, including a Proposed Amendment for discussion in an interview, is attached hereto as an Appendix.

IV. Status of Amendments

Amendments directed to the new ground of rejection raised in paragraph 6c and 96 of the Office Action of January 2005 are suggested in the Appendix. As noted below, Applicant believes that the claims meet all statutory requirements for definiteness. In the event that the arguments of § VI.B.5 and § VI.B.4, below are not accepted, Applicant proposes these amendments for entry by Examiner's Amendment, in order to reduce issues for appeal.

The amendments to the claims respond to concerns raised by the examiner that lack any statutory grounding. They are not made for a substantial reason related to patentability, and do not alter the scope of the claims.

V. Informal Summary of the Subject Matter

This § V presents a primer in the technology. The subject matter of specific claims is discussed in § VI.

The specification of this application teaches mechanisms for allowing an operating system, such as Microsoft Windows™, to be used with hardware for which that operating system was not specifically designed or tailored. For example, the specification teaches how to use Microsoft Windows – which is available only on Intel chips and a very few other processors – on a foreign RISC (Reduced Instruction Set Computer) processor. This allows a totally new hardware design to use a mature and widely-available operating system, even if the source code is not available to allow Windows to be tailored to or customized to that foreign RISC.

A. Background: Multi-Tasking and Context Switches

Since the early 1960's, computers have used "multi-tasking" to rapidly switch between two or more program processes, so that it appears that the computer is running several different programs simultaneously. For example, the computer on your desk allows you to run email, a word processor, and an internet browser simultaneously, and to switch between them rapidly. If one program is doing work that does not require user input (receiving email, printing a document, or downloading a web page, for example), the user can switch to an entirely different second program, and work there, while the first program continues to run.

A switch between programs, called a "context switch," is fairly analogous to switching between two paper-intensive tasks at your desk. When you switch tasks, you must carefully file away all of the papers of the old task and then retrieve the papers of the new task from their files, and lay them out on the desk for efficient access, and some record of what you were doing when you set the old task aside, so you know where to resume. Similarly, when a computer switches programs, the full status of the old program must be carefully stored: this typically includes the values of the registers, any descriptors that describe how the processor accesses the memory of the program, and a pointer to the point at which execution is to resume. Then, all of the corresponding information describing the new task (which was previously saved, when that task was switched out) must be loaded, so that control may be transferred to the new task.

Multi-tasking is typically implemented in a computer's operating system, typically at the lowest and most hardware-dependent layers of the operating system. For example, Exhibit 1 shows a very precise map for how an IBM mainframe computer stores every relevant piece of state in memory, and reloads new state from memory. Exhibit 2 shows the same kind of

information for Intel's Pentium chip as of 1996. Digital's Alpha has a third form. The three are necessarily entirely different, because the three have entirely different hardware resources.

Therefore, traditionally, an operating system is tied to one precise type of hardware. For example, when one installs a Microsoft operating system, one must know quite precisely what type of Intel chip is being used: there is a separate version of MS-DOS and of Windows for each new "major" generation of Intel chip. A mismatch will often result in a non-functioning system.

B. Background: the Chernoff '028 patent

Chernoff '028 provides a useful concrete example in which to introduce a number of relevant computer design concepts. Generally, Chernoff '028 is directed to allowing programs coded for a non-native computer family, for example, the Intel X86 family, to run on a computer of another family, for example, Digital Equipment Corp.'s Alpha RISC processor (col. 7, line 29).

The Office Action points to three totally unrelated portions of the Chernoff '028 patent. One section, col. 25 line 38 to col. 26 line 12, relates to multi-threading of X86 tasks, that is, changing from one task to another. The second, at col. 88, lines 22-40, relates to exception handling, that is, correcting problems that arise during execution, so that the same program can resume execution. The third, at col. 33, lines 1-9, relates to subroutine calls, which generally only incidentally raise an exception, and do not cause any change of execution thread.

1. Background: Multi-Threading

A number of programming systems provide a feature called "multi-threading." From a user's point of view, multi-threading creates the appearance that a single program is doing multiple things simultaneously. As a rule of thumb generalization, "multi-tasking" generally involves multiple programs, while "multi-threading" involves a single program. For example, most word processors allow a user to edit a document in the "foreground" while the document is being printed by the same program in the "background." http://en.wikipedia.org/wiki/Thread_%28computer_science%29 gives an overview:

> Many programming languages, operating systems, and other software development environments support what are called "**threads**" of execution. Threads are similar to processes, in that both represent a single sequence of instructions executed in parallel with other sequences, either by time slicing or

> multiprocessing. Threads are a way for a program to split itself into two or more simultaneously running tasks. (The name "thread" is by analogy with the way that a number of threads are interwoven to make a piece of fabric).
>
> A common use of threads is having one thread paying attention to the graphical user interface, while others do a long calculation in the background. As a result, the application more readily responds to user's interaction.

The advantage of "threads" is that communications and switching between two threads of a single program is more efficient than communications and switching between two entirely different programs. Because the same program remains in execution, the amount of state that needs to be saved and restored is much smaller than in a full process context switch.[1] For example, a context switch among threads might be analogous to switching from reading one document to reading another document, both documents relating to the same matter. One document need only be laid down on the desktop, and another picked up, without the need to file the entire matter away.

The basic implementation techniques are a simplified version of multi-tasking discussed in §V.A, above: the state of one thread is saved to memory, the state of another thread is loaded from memory into execution, hardware, and control is transferred to the new thread.

Though threads are less tied to particular hardware than are processes, traditionally, it has been impossible to use a thread manager for one family of computers (Intel X86/Pentium, for example) to manage threads of a foreign computer (Digital Equipment Corp.'s Alpha, discussed in Chernoff '028, or the Tapestry processor discussed in this application's specification). For example, Chernoff '028 at col. 25, lines 51-59 discusses saving and restoring the virtual X86 registers "... the condition code bits [and] copies of the integer registers EAX 104a, EBX 104b, ECS 104c, EDX 104d EDI 104e, ESI 104f, EBP 104g and ESP 104h." These are X86 hardware structures. Alpha uses different hardware structures (Chernoff '028, col. 2, lines 60-63), and there is simply no way to code access to them in the Alpha instruction set.

---

[1] The Wikipedia definition states that a context switch between processes is also a context switch between the threads of those processes.

2. Background: Software Simulators and the Chernoff '028 Patent

Some manufacturers of computer hardware have found it valuable to allow their computers to run software that is coded in a foreign instruction set. For example, the Chernoff '028 patent discusses a software system from Digital Equipment Corp. that allows Digital's Alpha processor to run Intel X86 application programs, using a "simulator" or an "interpreter." An interpreter is software that creates a "virtual" CPU of the foreign operating system.

An interpreter also has to provide many or most of the features of an operating system. For example, X86 programs cannot use the multi-tasking features of the Alpha operating system, because the Alpha operating system understands only Alpha hardware, not X86 hardware.

In order to support multi-threading capabilities in Intel X86 application programs, Chernoff's X86-to-Alpha interpreter system must provide a multi-threading capability totally separate from the multi-tasking and multi-threading capabilities that can be used to manage programs written for the Alpha. The context-switch capability of the Alpha operating system cannot context switch among Intel X86 threads, because, as noted above, thread managers are tied to particular families of hardware.

Chernoff '028 discusses a totally separate layer of multi-threading, only for X86 processes, at col. 25, line 36 to col. 26, line 21. Notably, all of the registers and structures, and "current state" mentioned in this section are X86 structures that are implemented as "virtual" software entities in the interpreter. This section of Chernoff '028 never discusses using the X86 multi-threading software to manage native Alpha processes, or switch between Alpha threads.

3. Background: Chernoff '028 and Exception Handling

At col. 88, lines 22-40, Chernoff '028 discusses exception handling. Certain kinds of events in a computer raise an "exception" or "interrupt," for example, divide-by-zero, an attempt to read or write an illegal memory location, a completion of a read or write to a disk, a timer expiration etc.

In most modern processors, the hardware classifies the exception into one of several categories and saves the state of the interrupted program. Then control is transferred to a "handler," software that further diagnoses the condition, and repairs it if possible. If the repair is successful, the handler may return control to the interrupted program, so that the program

resumes as if the exception had never occurred. If no repair is possible, then the program is aborted.

Chernoff '028 at col. 88, lines 22-40 discusses how exceptions that arise in the execution of Alpha instructions are routed to a native Alpha handler for resolution (col. 88, line 28), and exceptions that arise in the context of the execution of the virtual X86 are routed to an X86 CISC exception handler (col. 88, line 47).

Notably, this section of Chernoff '028 does not mention context switches. Nor does it mention handling an X86 exception in an Alpha handler, or vice-versa – at most, the mismatched handler determines that the exception must be handled elsewhere, and hands of the task of actually handling it.

C. The Embodiment in the Specification

The specification for this application discusses several techniques that allow the use of hardware and an operating system that are mismatched to each other.

Sections III and IV of the specification (pages 32-60) discuss mechanisms for allowing a non-native operating system, such as Microsoft Windows, to be used on hardware for which that operating system software has not been tailored. An introduction to this preferred embodiment appears at pp. 32-33 of the specification:[2]

> Referring to **Fig. 3a** and to Table 1, X86 threads (*e.g.*, **302**, **304**) managed by X86 operating system **306**, carry the normal X86 context, including the X86 registers, as represented in the low-order halves of r32-r55, the EFLAGS bits that affect execution of X86 instructions, the current segment registers, etc. In addition, if an X86 thread **302**, **304** calls native Tapestry libraries **308**, X86 thread **302**, **304** may embody a good deal of extended context, the portion of the Tapestry processor context beyond the content of the X86 architecture. A thread's extended context may include the various Tapestry processor registers, general registers r1-r31 and r56-r63, and the high-order halves of r32-r55 (see Table 1), the current value of ISA bit **194**...
>
> The Tapestry system manages an entire virtual X86 **310**, with all of its processes and threads, *e.g.*, **302**, **304**, as a single Tapestry process **311**. Tapestry operating system **312** can use conventional techniques for saving and restoring processor context, including ISA bit **194** of PSW **190**, on context switches

[2] This discussion, like all of § V, is a mere discussion of one concrete preferred embodiment, as a helpful aid to establish context for the discussion of the claims that follows. It is not a discussion of the claims themselves.

between Tapestry processes **311, 314**. However, for threads **302, 304** managed by an off-the-shelf X86 operating system **306** (such as Microsoft Windows or IBM OS/2) within virtual X86 process **311**, the Tapestry system performs some additional housekeeping on entry and exit to virtual X86 **310**, in order to save and restore the extended context, and to maintain the association between extended context information and threads **302, 304** managed by X86 operating system **306**. (Recall that Tapestry emulation manager **316** runs beneath X86 operating system **306**, and is therefore unaware of entities managed by X86 operating system **306**, such as processes and threads **302, 304**.)

Figs. 3a-3n describe the mechanism used to save and restore the full context of an X86 thread **304** (that is, a thread that is under management of X86 operating system **306**, and thus invisible to Tapestry operating system **312**) that is currently using Tapestry extended resources. In overview, this mechanism snapshots the full extended context into a memory location **355** that is architecturally invisible to virtual X86 **310**. A correspondence between the stored context memory location **355** and its X86 thread **304** is maintained by Tapestry operating system **312** and X86 emulator **316** in a manner that that does not require cooperation of X86 operating system **306**, so that the extended context will be restored when X86 operating system **306** resumes X86 thread **304**, even if X86 operating system **306** performs several context switches ... before the interrupted X86 thread **304** resumes. The X86 emulator **316** or Tapestry operating system **312** briefly gains control at each transition from X86 to Tapestry or back, including entries to and returns from X86 operating system **306**, to save the extended context and restore it at the appropriate time.

VI. Argument

A. Paragraph 5 of the Office Action - § 101 Issues

Paragraph 5 of the Office Action asserts that claim 5 is "directed to method steps which can be practiced mentally in conjunction with pen and paper. ... [It] is uncertain what performs each of the method steps. The examiner suggests [changing] to 'computer implemented methods' in the preamble." The Office Action errs on both the facts and the law.

Claim 5 recites "scheduling concurrent threads of control by a pre-existing thread scheduler of a computer." Claim 5 is "certain" that the method step is performed by software of "a computer," not mentally with pen and paper. A human being, even one with pencil and paper, cannot possibly perform this step.

Second, the Office Action suggests that recitation of a computer in the body of a claim may be ignored, and that it's the preamble that counts. This has never been the law. *Contrast In re Walter*, 618 F.2d 758, 767-70, 205 USPQ 397, 409 (CCPA 1980) (claim not patentable where

computer "signals" appears only in the preamble: "The specific end use recited in the preambles does not save the claims from the holding in *Flook*.") with *In re Johnson*, 589 F.2d 1070, 200 USPQ 199 (CCPA 1980) (a claim directed to essentially similar subject matter is patentable under § 101, because the "seismic trace" signal appears in the body of the claim).

Any § 101 rejection may be withdrawn.

B. Paragraph 4 of the Office Action - § 112 ¶ 2 Issues

MPEP § 2173.02 states the general test under § 112 ¶ 2 (emphasis added):

2173.02 Clarity and Precision

> The examiner's focus during examination of claims for compliance with the requirement for definiteness of 35 U.S.C. 112, second paragraph, is whether the claim meets the threshold requirements of clarity and precision, not whether more suitable language or modes of expression are available. ... he or she should allow claims which define the patentable subject matter with a reasonable degree of particularity and distinctness. Some latitude in the manner of expression and the aptness of terms should be permitted even though the claim language is not as precise as the examiner might desire. Examiners ... should not reject claims or insist on their own preferences if other modes of expression selected by applicants satisfy the statutory requirement.

Section 112, second paragraph issues must be considered under a "reasonableness" standard. For example, it is impermissible to rely on an artificially-constructed unreasonable or grammatically-incorrect reading of a claim when the most reasonable reading is grammatically correct. MPEP § 2173.02 continues:

> If the language of the claim is such that a person of ordinary skill in the art could not interpret the metes and bounds of the claim so as to understand how to avoid infringement, a rejection of the claim under 35 U.S.C. 112, second paragraph, would be appropriate. ... However, if the language used by applicant satisfies the statutory requirements of 35 U.S.C. 112, second paragraph, but the examiner merely wants the applicant to improve the clarity or precision of the language used, the claim must not be rejected under 35 U.S.C. 112, second paragraph, rather, the examiner should suggest improved language to the applicant.

1. Paragraph 4(a): "thread that are"

Paragraph 4(a) of the Office Action objects to the phrase "thread that are" as being "indefinite because it is grammatically incorrect and it is not made explicitly clear in the claim language whether there is a singular or plural amount of threads."

The last paragraph of claim 1 recites as follows:

> the entry exception, resumption exception, entry handler, and exit handler being cooperatively designed to maintain an association between one of the threads and an extended context of the thread through a context change induced by the operating system, the extended context including resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the operating system.

The claim is grammatically correct.

Using the normal rules of parsing English sentences and dividing them into their constituent clauses and prepositional phrases, the claim recites "resources of the computer ... that are beyond those resources whose association with the thread is maintained by the operating system." There is proper singular/plural agreement between "resources" and "are."

The underlined instance of "with the thread" is a separate prepositional phrase. No word of the claim requires singular/plural agreement with this instance of "thread."

It is improper for an Office Action to artificially extract three words out of context as a fragment, and then reject that fragment as "grammatically incorrect." It is also improper to "reject" a claim under § 112 ¶ 2 with no showing that "a person of ordinary skill in the art could not interpret the metes and bounds" required by MPEP § 2173.02. Under any reasonable parsing of the claim, it is grammatically correct. Because the Office Action makes no allegation that the one of ordinary skill would have difficulty understanding the scope of the claim, no rejection is even raised.

If the Examiner wishes to "suggest improved language," for non-statutory reasons of personal preference, Applicant will entertain such suggestions. However, no rejection properly lies.

2. Paragraph 4(b), first half: "extended context" and "modified context"

Paragraph 4(b) states that "the term 'extended context' (line 19 [of claim 1]) is indefinite because it is not made explicitly clear whether this term relates to the 'modified context' (line 6) or if it introduces a new type of context."

The claim is drafted using normal claim drafting rules: when different words are used, they refer to "new" things:

- Claim 1 explicitly recites that the "modified context" is the context modified by "the entry handler programmed to ... modify the thread context before delivering the modified context to the operating system."
- Claim 1 then explicitly recites that the "extended context" includes "resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the operating system."

The two claim limitations relate to each to exactly the extent recited in the claim. They may overlap to some extent, or they may not. The claim is deliberately broad in this respect. "Breadth is not indefiniteness." MPEP § 2173.04. The claim is drafted in reliance on this well-established principle of law, and the Office Action sets forth no reason to believe that § 2173.04 does not apply to this claim.

If the Examiner perceives any ambiguity, Applicant requests a more precise explanation, that lays out some reason to believe that the claim does not mean what it says. Applicant also requests a citation to some authority that sets out the legal standard applied.

3. Paragraph 4(b), second half

The second half of paragraph 4(b) is directed to language that does not exist in any claim. No rejection is raised.

4. Paragraph 4(c): "without modifying a pre-existing operating system"

Applicant respectfully observes that "without modifying a pre-existing operating system" is not ambiguous.

Each claim simply recites that certain results are achieved without modifying the operating system or thread handler, as the case may be, to achieve that result. For example, claim 5 (even without entry of the proposed amendment) recites as follows:

> without modifying the thread scheduler, maintaining an association between one of the threads and an extended context of the thread through a context change induced by the thread scheduler, the extended context including resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the thread scheduler.

For example, this language would cover a process that allows Microsoft Windows to perform context switches among threads of a RISC computer, even though there was no modification of Windows source code.

The claims are "reasonably precise," and "as precise as the subject matter allows," and thereby meets the requirements of MPEP § 2173.02. As a practical matter, "establishing an entry exception," "establishing a resumption exception," and "maintaining an association between one of the threads and an extended context of the thread" almost certainly require creation or modification of some data structures, possibly associated with the operating system. They almost certainly require some code outside the operating system. However, the claims unambiguously recite that there is no modification of the "operating system" or "thread scheduler" itself associated with the result recited in the rest of the claim. Thus, these situations unambiguously fall within the respective claims. The Office Action offers no suggestion or showing that one of ordinary skill could read the claim any other way, or would have any difficulty doing so.

The Office Action discusses four separate cases, none of which identify any ambiguity in any claim.

As the Office Action itself concedes, modifying "associated data registers" is a change to something "associated" with an operating system, not a change to the operating system itself. This suggestion in the Office Action is irrelevant.

Any difference between modifying the "code" or "functions" is illusory. For example, changing the code results in a change to function. Changing function, as a practical matter, requires changing the code, and any change to the code with result in a change in function. Applying the "reasonableness" threshold for § 112 ¶ 2, there is no reason to differentiate between these cases: as a practical matter, all of them or none of them change simultaneously. There is no ambiguity between these cases.

Applicant does not understand what might be meant by changing "configuration" of an operating system. Perhaps the reference is to replacing an operating system with another that has been modified for use with a different hardware configuration? This would unambiguously be "modifying the operating system." Perhaps it is to modifying data structures associated with the operating system? This is unambiguously modifying only something "associated" with the operating system, not the operating system itself.

In sum, the Office Action proposes no set of facts that would present any difficulty in "understand[ing] how to avoid infringement." Without that showing, the claim is not indefinite.

If the Examiner wishes to "suggest improved language" that does not alter the scope of the claims, for non-statutory reasons of personal preference, Applicant will entertain such suggestions. In the alternative, language is proposed for entry by Examiner's Amendment.

5. Paragraph 4(d): "returning control"

Paragraph 4(d) questions the "antecedent basis" for "returning control," even though neither the word "said" nor the word "the" is used.

MPEP § 2173.05(e) makes clear that "antecedent basis" is not an independent or automatic ground of rejection; there must be some showing that the claim is unclear to one of ordinary skill:

> **2173.05(e) Lack of Antecedent Basis**
> A claim is indefinite when it contains words or phrases whose meaning is unclear. ... Obviously, however, the failure to provide explicit antecedent basis for terms does not always render a claim indefinite. If the scope of a claim would be reasonably ascertainable by those skilled in the art, then the claim is not indefinite. ... Inherent components of elements recited have antecedent basis in the recitation of the components themselves. ... [T]he limitation "the outer surface of said sphere" would not require an antecedent recitation that the sphere has an outer surface.

"Returning control" is a well-established term of art that requires no definition or antecedent basis to be understood by those of ordinary skill. For example, claim 6 of U.S. Patent No. 6,826,675 and claim 1 of U.S. Pat. No. 6,711,644 use "returning control" in almost the same way as these claims. "Returning control"[3] is used in hundreds of computer patents – it is a term of art that requires no antecedent basis.

---

[3] Or similar language like "return of control" or "control is returned."

Further, the existing claim language of claim 33, "scheduling concurrent threads of control by the operating system" provides antecedent basis for "control." Claim 79 recites "returning control to a caller of the service routine:" because those in the art inherently associate a "caller" with a "return," the "caller" inherently provides antecedent basis for the return. Both claims thus provide "antecedent basis" pursuant to MPEP § 2173.05(e).

No rejection is raised or warranted. Nonetheless, to reduce issues for appeal, Applicant offers proposed language for entry by Examiner's Amendment. Applicant will entertain any "suggestion for "improved language" based on the examiner's non-statutory personal preferences.

C. Claims 1-4, 5-32, 82, 83

1. Claims 1-4, 5-32, 82 and 83 are Patentable on the Merits

Claim 5 is discussed in connection with Nilsen '665 and Chernoff '028 at paragraphs 6, 8, and 97-100 of the Office Action. Claim 5 recites as follows:

> 5. A method, comprising:
>
> scheduling concurrent threads of control by a pre-existing thread scheduler of a computer, each thread having an associated context, an association between a thread and a set of computer resources of the associated context being maintained by the thread scheduler; and
>
> without modifying the thread scheduler, maintaining an association between one of the threads and an extended context of the thread through a context change induced by the thread scheduler, the extended context including resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the thread scheduler.

Claim 5 recites an "extended context" that includes "resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the thread scheduler." The Office Action suggests that this might be met by Chernoff '082. The Office Action is incorrect.

The first error appears in paragraph 97, which equates "thread schedulers" with "handlers." The two things are quite different. A "thread scheduler" is usually designed to place a thread to execution in exactly the same state[4] in which the thread was suspended; in contrast, a

[4] Or, possibly, as modified by something other than the thread scheduler itself.

handler is almost always designed to change the thread's state by removing the condition that raised the exception.

The second error is one of omission: the Office Action never identifies any resource in either reference that is a "resource beyond those resources whose association with the thread is maintained by the thread scheduler." When an Office Action is simply silent on a claim limitation, it raises no rejection at all. Claim 5 is not rejected.

The third error is at the end of paragraph 6, in the statement "It would have been obvious to ... because it allows the handlers to have control over the context data structure stored in the table." Chernoff's "table" stores "pointers to methods," Chernoff '028, claim 1, and col. 3, line 55, not to the context data structures.

The fourth error is failing to recognize that Chernoff '028 manages his X86 threads and his Alpha threads in two completely separate ways. The Office Action focuses most on col. 25, line 38 to col. 26 line 12 of Chernoff '082, discussing the "Context Data Structure" 180. Chernoff '028 teaches that his context data structure holds only X86 contexts (col. 25, lines 50-65). Chernoff '028 only discusses using this multitasking manager to manage X86 processes (col. 25, lines 41-42). Thus, Chernoff's multitasking manager and context data structure discussed here are exactly tailored to each other – the manager only manages X86 context data, and the context data structure only holds X86 context data. Chernoff's "context data structure" never stores a "resource beyond those resources whose association with the thread is maintained by" Chernoff's multitasking manager of col. 25.

Alpha context is not mentioned in this portion of Chernoff '028. Indeed, it would be quite impossible to store or switch Alpha context using the mechanisms described in col. 25-26. For example, the Alpha processor has 31 integer registers and 31 floating-point registers that must be stored on context switch, each 64 bits wide (see the "Compaq Computer Corp., Compiler Writer's Guide for the Alpha 21264 (1999)," at pages 3-9 to 3-11, of record in this application, or at `//ftp.digital.com/pub/Digital/info/semiconductor/literature/cmpwrgd.pdf`). Alpha's data registers alone require almost 4000 bits of storage. In contrast, Chernoff '028 teaches that his X86 "context data structure" only has space for the X86's eight

32-bit integer registers (col. 25, lines 56-59; col. 16, lines 32-33), a total of only 256 bits. 4000 bits of information cannot fit in 256 bits of storage.[5]

The remaining portions of Chernoff '028 designated in the Office Action are unrelated to each other, or to the "extended context" recited in claim 5. For example, col. 88, lines 22-40 never mentions the "context data structure 180" or its associated multitasking manager. Instead, col. 88 discusses a fairly conventional exception mechanism that stores Alpha context of an Alpha program in the conventional way (an Alpha context save, not the X86 context save discussed in col. 25-26), dispatches to an Alpha handler, and handles Alpha exceptions in the conventional way. Even the residual X86 case, col. 88, lines 45-52, has no relationship to the X86 mechanisms discussed in col. 25-26: the exception is simply handed off to the CISC exception handler, without saving context into the "context data structure" 180. The Office Action does not suggest that this portion of Chernoff '028 mentions any problem that would call for an "extended context" as recited in claim 5, and none is apparent.

The Office Action raises a new ground of rejection, relating to col. 33, lines 1-9 of Chernoff '028. Col. 33 is even further afield. This portion relates to simple subroutine calls, in this case from a portion of the program that has been binary translated from X86 to Alpha instructions (col. 32, lines 63-64) to one that is still in pure X86 code to be executed in the interpreter. Typical subroutine calls do not cause an interrupt, a context save, or a context switch. Nothing in this portion of Chernoff '028 suggests anything atypical in any of these respects that would suggest any interaction with the mechanisms of col. 25-26 or col. 88.

The Office Action itself concedes that Nilsen '665 teaches nothing corresponding to this claim language.

Because claim 5 recites a feature that is absent from both Nilsen '665 and Chernoff '028, claim 5 is patentable over this combination.

---

[5] Both processors have a number of control registers that must also be stored; Alpha's set is larger than the storage provided for them in Chernoff '028.

2. Procedurally, the Office Action Fails to Raise any Rejection Under § 103

The Office Action fails to comply with the procedural minima set out in 37 C.F.R. § 1.104(c)(2) and MPEP § 2141-2144.09 necessary to raise a rejection. Until these minima have been met, no *prima facie* rejection even exists.

The Office Action simply designates a number of unrelated portions of Chernoff '028, without explaining the "pertinence" of the indicated portions. This fails to comply with 37 C.F.R. § 1.104(c)(2), which requires both the designation of the "particular portions relied on" and a clear explanation of "pertinence" of those portions.

The Office Action merely considers a few isolated words of the claim, with no attention to the words of the claim that interconnect those isolated words. The analysis in the Office Action is too incomplete to constitute a rejection.

The Office Action asserts that Nilsen's handlers would find some unspecified benefit in having control over Chernoff's "context data structure," but cites no prior art to support this statement. Statements of "motivation to combine" must be supported by substantial evidence.[6]

The Office Action makes no showing that the three distantly-separated portions of Chernoff '028 have any relationship to each other, and no showing of "motivation to combine" them. Because the Office Action fails to comply with MPEP § 2143.01, it fails to raise a rejection.

The Office Action makes no showing of "reasonable expectation of success," as required by MPEP § 2143.02.[7]

---

[6] MPEP § 2143.03 restates Federal Circuit law: *every* limitation of a claim must be met by prior art drawn from one of the categories of § 102. No limitation may be rejected based on bald assertion. *In re Zurko*, 258 F.3d 1379, 1385, 59 USPQ2d 1693, 1697 (Fed. Cir. 2001) ("the Board cannot simply reach conclusions based on its own understanding or experience—or on its assessment of what would be basic knowledge or common sense. Rather, the Board must point to some concrete evidence in the record in support of these findings."); *In re Lee*, 277 F.3d 1338, 1343-44, 61 USPQ2d 1430, 1434 (Fed. Cir. 2002) ("'common knowledge and common sense' on which the Board relied in rejecting Lee's application are not the specialized knowledge and expertise contemplated by the Administrative Procedure Act. Conclusory statements such as those here provided do not fulfill the agency's obligation."); *Motorola v. Interdigital Technology Corp.*, 121 F.3d 1461, 1466-67, 43 USPQ2d 1490, 1490-91 (Fed. Cir. 1997) (every element must be met by prior art, even elements that are "well known" standing alone). If DiBrino is not offered as a "new ground of rejection," then no rejection exists at all.

The Office Action concedes that no reference corresponds to "maintaining an association between one of the threads and an extended context of the thread through a context change induced by the thread scheduler," and does not allege that this element is "capable of instant and unquestionable demonstration as being well-known." Naked assertion is not an evidentiary basis permitted by MPEP § 2144-2144.09. Without some citation to a permissible class of evidence, the Office Action fails to comply with MPEP § 2143.03, and no rejection exists.[8]

Without the showings required by MPEP §§ 2143.01 and 2143.02, no *prima facie* rejection exists.

3. Claims 1-4, 6-32, 46, 54, 57, 58, 76-78, 82, and 83 are patentable with Claim 5

Claim 1 recites similar language, a "pre-existing operating system" that works with "extended context" in a similar way. Claim 1 is patentable for similar reasons.

Claims 2-4, 6-32, 46, 54, 57, 58, 76-78, 82, and 83 are dependent on claims 1 or 5, or recite language similar to that discussed above, and patentable therewith.

D. Claims 33-55 and 56-78

Claim 33 is mentioned in paragraphs 6 and 25 of the Office Action. Claim 33 recites as follows (though may be amended, without altering its scope, if the Proposed Amendment is entered):

> 33. A method, comprising:
>
> establishing an entry exception to be raised on each entry to a computer operating system at a specified entry point or on a specified condition;
>
> establishing a resumption exception to be raised on each resumption from the operating system complementary to one of the specified entries;
>
> on detecting a specified entry to the operating system from an interrupted process of the computer, raising and servicing the entry exception, and then

---

[7] Paragraph 98 of the Office Action misparaphrases the argument and the MPEP test for obviousness. "Motivation to combine" and "reasonable expectation of success" are two distinct elements – one does not show one in order to establish the other. MPEP §§ 2143, 2143.01 and 2143.02.

[8] To the degree that "maintaining an association between one of the threads and an extended context of the thread through a context change induced by the thread scheduler" and having control over Chernoff's "context data structure" involve official notice, applicant calls for a reference or an affidavit pursuant to 37 C.F.R. § 1.104(d)(2).

entering the operating system to perform a service associated with the original operating system entry; and

on detecting a complementary resumption, raising and servicing the resumption exception, and returning control to the interrupted process.

Claim 33 recites wrapping a conventional or pre-existing entry point of a computer operating system in a pair of additional exceptions: one to be raised on entry, one to be raised on resumption. One example embodiment can be seen by following arrows (7), (8), (9), (12), (13) and (14) of Fig. 3a. In this example, when an interrupt occurs (388 of Fig. 3a), normally execution would trap into the operating system (306 of Fig. 3a). Instead, a second exception (arrow (7) of Fig. 3a, corresponding to the "entry exception" of claim 33) is raised. This causes control to be transferred to a handler (350 of Fig. 3a, corresponding to "servicing the entry exception" of claim 33). When that handler returns control (arrow (9) of Fig. 3a), typically the operating system handles the original interrupt. When the operating system executes its return of control instruction (arrow (12) of Fig. 3a, corresponding to "a complementary resumption" of claim 33), another exception (arrow (13) of Fig. 3a, corresponding to the "resumption exception" of claim 33) is raised. This exception transfers control to a third handler (lower right corner of Fig. 3a, "servicing the resumption exception" of claim 33). The resumption exception handler returns control to the original thread (arrow (14) of Fig. 3a, corresponding to the "returning control to the interrupted process" of claim 33). Thus, a typical embodiment of claim 33 would involve three exceptions and associated handlers. See Figs. 3h, 3i and 3j.[9]

Paragraph 6 of the Office Action juxtaposes the "resumption exception to be raised on each resumption from the operating system complementary to one of the specified entries" to Nilsen '665, col. 30, lines 25-28, col. 33, lines 40-67, col. 37, lines 60-67, and col. 25, lines 40-67). The Office Action merely designates these portions of Nilsen '665, without identifying any "resumption exception" being raised there, or otherwise explaining the pertinence.[10] Nilsen '665 discusses nothing corresponding to the "resumption exception." The designated portions at best discuss an implementation of the setjmp/longjmp feature of the standard "C" library, for

[9] This comparison of the specification to the claim is not a limiting description of the invention or the claim; it is merely offered as a concrete preferred embodiment to assist in understanding the claim.

[10] Because the Office Action fails to comply with the requirements of 37 C.F.R. § 1.104(b)(2) for setting out a rejection, no rejection has been raised.

handling a single exception (See Exhibit 4). These portions of Nilsen '665 never suggest that a "resumption exception" is raised, separate from the condition that triggered the setjmp/longjmp itself. Any particular use of setjmp/longjmp involves only one exception and one handler, and nothing corresponding to the "resumption exception" following the setjmp/longjmp, as required by claim 33.[11]

The Office Action does not suggest that Chernoff '028 teaches anything corresponding to the "resumption exception," and a brief review of Chernoff '028 reveals no such teaching.

Claim 33 recites a limitation absent from all references. Claim 33 is therefore patentable. Independent claims 1 and 56 recites similar language and are patentable for similar reasons. Dependent claims 2-4, 6-14, 34-55, and 57-78 either recite similar language, or are dependent on these claims, and are therefore patentable.

For the same reasons discussed above in § VI.C.2, the Office Action is procedurally inadequate to raise a rejection.

E. Claim 79

Claim 79 is discussed in paragraphs 6, 55 and 102 of the Office Action. Claim 79 recites as follows:

> 79. A method, comprising:
>
> during invocation of a service routine of a computer, passing a linkage return address to the service routine at which to resume execution on completion of the service, the linkage return address being deliberately chosen so that an attempt to execute an instruction from the linkage return address on return from the service routine will raise a program execution exception;
>
> on return from the service routine, attempting to execute the instruction at the linkage return address and raising the chosen exception; and
>
> after servicing the exception, returning control to a caller of the service routine.

Claim 79 recites "the linkage return address being deliberately chosen so that an attempt to execute an instruction from the linkage return address on return from the service routine will

---

[11] MPEP § 713.04 provides no basis for giving an applicant's interview summary less weight than the examiner's summary. Applicant's paper of July 3, 2003 was filed one day after the interview it records. The Examiner's remarks of January 2005 are 18 months after the fact. Applicant's interview summary is therefore entitled to weight as an accurate recordation of the conversation.

raise a program execution exception." One example is discussed at § IV.H, at page 49-50 of the specification.

The Office Action itself admits that that nothing in either Nilsen '665 or Chernoff '028 corresponds to the underlined language. Paragraph 102 of the January 2005 Office Action raises a new ground of rejection, based on the Abstract of DiBrino '894.[12] But DiBrino's abstract demonstrates the absence of the underlined claim language from the prior art. For example, DiBrino's abstract never mentions a "return from a service routine."[13] This limitation is absent from the prior art, and the claim therefore cannot be obvious. MPEP § 2143.03.

The Office Action states that the proposed combination would "provide for control for exceptions to occur," but identifies no respect in which the control over exceptions in Nilsen '665 or Chernoff '028 is inadequate, how the supposed control would be exercised, or any respect in which the combination Nilsen '665 and Chernoff '028 offers any benefit over the two standing separately. A showing of "motivation to modify" must be more than "statements of generalized advantages and convenient assumptions." *In re Beasley*, 117 Fed.Appx. 739, 744 (Fed. Cir. 2004). No credible motivation to modify or combine is shown.

For these reasons, claim 79 is patentable over the art. Claims 80-81 are dependent thereon, and allowable therewith. Claims 52-53 recite similar language, and are patentable for similar reasons.

VII. Conclusion

In view of these remarks, Applicant respectfully submits that the claims are in condition for allowance. Applicant requests that the application be passed to issue in due course. In the even that the "returning control" rejection is maintained, Applicant requests entry of the accompanying proposed amendment for appeal. The Examiner is urged to telephone Applicant's

---

[12] See footnote 6.

[13] Further reliance on "official notice" is impermissible for two reasons. First, MPEP § 2144.03(A) cautions "It would not be appropriate for the examiner to take official notice of facts without citing a prior art reference where the facts asserted to be well known are not capable of instant and unquestionable demonstration as being well-known." The clear failure of DiBrino to supply all of the "official notice" subject matter shows that the original assertion of "official notice was incorrect." Second, "It is never appropriate to rely solely on 'common knowledge' in the art without evidentiary support in the record, as the principal evidence upon which a rejection was based."

undersigned counsel at the number noted below if it will advance the prosecution of this application, or with any suggestion to resolve any condition that would impede allowance. In the event that any extension of time is required, Applicant petitions for that extension of time required to make this response timely. Kindly charge any additional fee, or credit any surplus, to Deposit Account No. 23-2405, Order No. 114596-05-4013.

Respectfully submitted,

WILLKIE FARR & GALLAGHER LLP

Dated: March 21, 2005

By: _____

David E. Boundy
Registration No. 36,461

WILLKIE FARR & GALLAGHER LLP
787 Seventh Ave.
New York, New York 10019
(212) 728-8000
(212) 728-8111 Fax

Claims Appendix to

Response to Office Action

PROPOSED AMENDMENT

1. (currently amended) A method, comprising:

without modifying source code of a pre-existing operating system of the computer, establishing an entry exception to be raised on each entry to the operating system at a specified entry point or on a specified condition, the entry exception having an associated entry handler, the entry handler programmed to save a context of a thread whose execution is [[an]] interrupted ~~thread~~ and to modify the thread context before delivering the modified context to the operating system;

without modifying source code of the operating system, establishing a resumption exception to be raised on each resumption from the operating system complementary to one of the specified entries, the resumption exception having an associated exit handler, the exit handler programmed to restore the context saved by a corresponding execution of the entry handler;

scheduling concurrent threads of control by the operating system, each thread having an associated context, the association between a thread and a set of computer resources of the associated context being maintained by the operating system;

on detecting a specified entry to the operating system from an interrupted thread of the computer, raising and servicing the entry exception; and

on detecting a complementary resumption, raising and servicing the resumption exception, and resuming execution of ~~returning control to~~ the interrupted thread;

the entry exception, resumption exception, entry handler, and exit handler being cooperatively designed to maintain an association between one of the threads and an extended context of the thread through a context change induced by the operating system~~, the extended context including resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the operating system~~.

2. (original) The method of claim 1, wherein the operating system is an operating system for a computer architecture other than the architecture native to the computer.

3. (original) The method of claim 1, wherein the operating-system-maintained resources of the thread context include data registers of the non-native computer architecture, the method further comprising:

modifying at least half of the data registers of the portion of the thread context maintained by the operating system before delivering the thread to the non-native operating system.

4. (original) The method of claim 1, wherein thread scheduler and the thread execute in different instruction sets of the computer, and the entry and exit exception are automatically invoked, without explicit software request, on a transition between the thread instruction set and the operating system instruction set.

5. (previously presented) A method, comprising:

scheduling concurrent threads of control by a pre-existing thread scheduler of a computer, each thread having an associated context, an association between a thread and a set of computer resources of the associated context being maintained by the thread scheduler; and

without modifying source code of the thread scheduler to address resources of an extended thread context, the extended context including resources of the computer associated with a thread that are beyond those resources whose association with the thread is maintained by the thread scheduler, maintaining an association between one of the threads and an extended context of the thread through a context change induced by the thread scheduler~~, the extended context including resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the thread scheduler~~.

6. (currently amended) The method of claim 5, wherein the thread scheduler is a component of an operating system of the computer, and further comprising:

establishing an entry exception to be raised on each entry to the operating system at a specified entry point or on a specified condition;

establishing a resumption exception to be raised on a resumption from the operating system following on a specified entry;

on detecting a specified entry to the operating system from a process of the computer whose execution is ~~an~~ interrupted ~~process of the computer~~, raising the entry exception, and establishing the association as part of servicing the entry exception; and

raising the resumption exception, and as part of servicing the resumption exception, reestablishing the context in association with the resumed thread, and resuming execution of ~~returning control to~~ the interrupted process.

7. (original) The method of claim 6, wherein an exception handler for the entry exception is programmed to save a context of the interrupted process and modify the thread context before delivering the modified context to the operating system; and

an exception handler for the resumption exception is programmed to restore the context saved by a corresponding execution of the entry exception handler.

8. (original) The method of claim 7, wherein the operating system is an operating system for a computer architecture other than the architecture native to the computer.

9. (original) The method of claim 8, wherein the computer additionally executes an operating system native to the computer, and each exception is classified for handling by one of the two operating systems.

10. (original) The method of claim 8, wherein operating system and the interrupted thread execute in different instruction set architectures of the computer.

11. (original) The method of claim 6, wherein the operating system is in a binary code for a computer architecture non-native to the architecture of the computer.

12. (original) The method of claim 11, wherein the computer additionally executes an operating system native to the computer, and each exception is classified for handling by one of the two operating systems.

13. (original) The method of claim 11, wherein operating system and the interrupted thread execute in different instruction set architectures of the computer.

14. (original) The method of claim 11, wherein the resources of the context maintained in association with the thread by the non-native operating system include data registers of the non-native computer architecture, the method further comprising:

in the entry exception handler, modifying at least half of the data registers of the portion of the process context maintained by the non-native operating system before delivering the process to the non-native operating system, at least some of the modified registers being redundantly written with data to enable checking of the validity of the contents of the context in the resumption exception handler.

15. (original) The method of claim 6, wherein thread scheduler and the thread execute in different execution modes of the computer, and the steps to maintain the association between the thread and the context are automatically invoked, without explicit software request, on a transition between the thread execution mode and the thread scheduler execution mode.

16. (original) The method of claim 15, wherein the thread execution mode and the thread scheduler execution mode are two different instruction set architectures of the computer.

17. (original) The method of claim 6, further comprising:

during servicing the entry exception, saving a portion of the context of the computer, and altering the context of an interrupted thread before delivering the interrupted thread and its corresponding context to the operating system.

18. (original) The method of claim 6, further comprising the step of modifying a linkage return address for resumption of the thread to include information used to maintain the association.

19. (original) The method of claim 18, wherein the modification leaves at least half of the bits of the linkage return address intact.

20. (original) The method of claim 5, wherein the thread scheduler is an operating system for a computer architecture other than the architecture native to the computer.

21. (original) The method of claim 20, wherein the computer additionally executes an operating system native to the computer, and each exception is classified for handling by one of the two operating systems.

22. (original) The method of claim 20, wherein operating system and the interrupted thread execute in different instruction set architectures of the computer.

23. (original) The method of claim 5, wherein thread scheduler and the thread execute in different execution modes of the computer, and the steps to maintain the association between the thread and the context are automatically invoked, without explicit software request, on a transition between the thread execution mode and the thread scheduler execution mode.

24. (original) The method of claim 23, wherein the thread execution mode and the thread scheduler execution mode are two different instruction set architectures of the computer.

25. (original) The method of claim 23, further comprising the step of setting of a register to a value that specifies actions to be taken by an exception handler invoked on the transition to

convert operands from one form to another to conform to a data storage convention of the thread scheduler execution mode.

26. (original) The method of claim 5, further comprising:

in an interrupt handler of the computer, saving a portion of the context of the computer, and altering the context of an interrupted thread before delivering the interrupted thread and its corresponding context to the thread scheduler.

27. (original) The method of claim 20, wherein the operating-system-maintained resources of the thread context include data registers of the non-native computer architecture, the method further comprising:

modifying at least half of the data registers of the portion of the thread context maintained by the operating system before delivering the thread to the non-native operating system.

28. (original) The method of claim 27, wherein at least some of the modified registers are overwritten by a timestamp.

29. (original) The method of claim 27, wherein at least some of the modified registers are overwritten by information indicating a storage location at which at least the portion of the thread context to be modified is saved before the modifying.

30. (original) The method of claim 5, further comprising the step of modifying a linkage return address for the thread to include information used to maintain the association.

31. (original) The method of claim 30, wherein the modification leaves at least half of the bits of the linkage return address intact.

32. (previously presented) The method of claim 5, further comprising either the step of deferring delivery of an interrupt before interrupting the thread by a time sufficient to allow the thread to reach a checkpoint, or the step of rolling execution of the thread back to a checkpoint, the checkpoints being points in the execution of the thread where the amount of extended context, being the resources of the thread that are beyond those whose resource association with the thread is maintained by the thread scheduler, is reduced.

33. (currently amended) A method, comprising:

establishing an entry exception to be raised on each entry to a computer operating system at a specified entry point or on a specified condition;

establishing a resumption exception to be raised on each resumption from the operating system complementary to one of the specified entries;

on detecting a specified entry to the operating system from interruption of a ~~an interrupted~~ process executing on [[of]] the computer, raising and servicing the entry exception, and then entering the operating system to perform a service associated with the original operating system entry; and

on detecting a complementary resumption, raising and servicing the resumption exception, and resuming execution of ~~returning control to~~ the interrupted process.

34. (original) The method of claim 33, wherein an exception handler for the entry exception is programmed to save a context of the interrupted process and modify the thread context before delivering the modified context to the operating system; and

an exception handler for the resumption exception is programmed to restore the context saved by a corresponding execution of the entry exception handler.

35. (original) The method of claim 34, wherein the operating system is an operating system for a computer architecture other than the architecture native to the computer.

36. (original) The method of claim 35, wherein operating system and the interrupted thread execute in different instruction set architectures of the computer.

37. (original) The method of claim 35, wherein the resources of the context maintained in association with the thread by the operating system include data registers of the non-native computer architecture, the method further comprising:

in the entry exception handler, modifying at least half of the data registers of the portion of the process context maintained by the operating system before delivering the process to the non-native operating system, at least some of the modified registers being redundantly written with data to enable checking of the validity of the contents of the context in the resumption exception handler.

38. (original) The method of claim 34, wherein the operating system and the process execute in two different instruction set architectures of the computer, and at least some of the steps to maintain the association between the process and the context are automatically invoked, without explicit software request, on a transition between the instruction set architectures.

39. (original) The method of claim 34, further comprising the step of modifying a linkage return address for the process to include information used to restore the context.

40. (original) The method of claim 33, wherein the operating system is an operating system for a computer architecture other than the architecture native to the computer, unmodified for execution on the computer.

41. (original) The method of claim 40, wherein the computer additionally executes an operating system native to the computer, and each exception is classified for handling by one of the two operating systems.

42. (original) The method of claim 40, wherein operating system and the interrupted thread execute in different instruction set architectures of the computer.

43. (original) The method of claim 33, wherein the operating system and the process execute in different execution modes of the computer, and the steps to maintain the association between the process and the context are automatically invoked, without explicit software request, on a transition between the process execution mode and the operating system execution mode.

44. (original) The method of claim 43, wherein the process execution mode and the operating system execution mode are two different instruction set architectures of the computer.

45. (original) The method of claim 33, wherein a service routine for the entry exception modifies at least half of the data registers of the portion of a process context maintained in association with the process by the operating system before delivering the process to the non-native operating system.

46. (previously presented) The method of claim 45, wherein at least some of the modified registers are overwritten by information indicating a storage location at which at least the extended context, the extended context being the resources beyond those whose resource association with the process is maintained by the operating system, is saved before the modifying.

47. (original) The method of claim 46, wherein at least some of the modified registers are overwritten by a value that enables validation of the contents of the context.

48. (original) The method of claim 45, wherein at least some of the modified registers are overwritten by a value that enables validation of the contents of the context.

49. (original) The method of claim 45, wherein at least some of the modified registers are overwritten by a timestamp.

50. (original) The method of claim 33, further comprising the step of modifying a linkage return address for the process to include information used to maintain the association.

51. (original) The method of claim 50, wherein the modification leaves at least half of the bits of the linkage return address intact.

52. (original) The method of claim 33, further comprising:

as part of servicing the entry exception, modifying a linkage return address of the interrupted process, the return address being deliberately chosen so that an attempt to execute an instruction from the return address on return from the operating system will raise the resumption exception.

53. (original) The method of claim 52, wherein the linkage return address is selected to point to a memory page having a memory attribute that raises the chosen exception on at attempt to execute an instruction from the page.

54. (previously presented) The method of claim 33, further comprising either the step of rolling execution of the process back to a checkpoint in the execution of the process where the amount of extended context, the extended context being the resources of the process context beyond those whose resource association with the process is maintained by the operating system, is reduced.

55. (previously presented) The method of claim 33, further comprising either the step of deferring delivery of an interrupt before interrupting the process by a time sufficient to allow the process to reach a checkpoint in the execution of the process where the amount of extended

context, the extended context being the resources of the process context beyond those whose resource association with the process is maintained by the operating system, is reduced.

56. (original) A method, comprising:

without modifying source code of a pre-existing operating system of the computer, establishing an entry handler for execution at a specified entry point or on a specified entry condition to the operating system, the entry handler programmed to save a context of an interrupted thread and modify the thread context before delivering the modified context to the operating system;

without modifying source code of the operating system, establishing an exit handler for execution on resumption from the operating system following an entry through the entry handler, the exit handler programmed to restore the context saved by a corresponding execution of the entry handler.

57. (previously presented) The method of claim 56, further comprising:

scheduling concurrent threads of control by the operating system, each thread having an associated context, an association between a thread and a set of computer resources of the associated context being maintained by the operating system; and

the entry and exit handlers being programmed to maintain an association between one of the threads and an extended context of the thread through a context change induced by the operating system, the extended context including resources of the computer associated with the thread that are beyond those resources whose association with the thread is maintained by the operating system.

58. (original) The method of claim 57, wherein the operating system is an operating system for a computer architecture other than the architecture native to the computer.

59. (original) The method of claim 57, wherein the operating system and the thread execute in different execution modes of the computer, and the steps to maintain the association

between the thread and the context are automatically invoked, without explicit software request, on a transition between the thread execution mode and the operating system execution mode.

60. (original) The method of claim 57, further comprising:

in the entry handler, saving a portion of the context of the computer, and altering the context of the interrupted thread before delivering the interrupted thread and its corresponding context to the operating system.

61. (original) The method of claim 57, wherein the entry handler alters at least half of the data registers of the portion of a thread context maintained in association with the thread by the operating system before delivering the thread to the operating system.

62. (original) The method of claim 57, further comprising the step of modifying a linkage return address for the thread to include information used to maintain the association.

63. (original) The method of claim 56, wherein the operating system is an operating system for a computer architecture other than the architecture native to the computer.

64. (original) The method of claim 63, wherein the computer additionally executes an operating system native to the computer, and each interrupt or exception is classified for handling by one of the two operating systems.

65. (original) The method of claim 63, wherein operating system and the interrupted thread execute in different instruction set architectures of the computer.

66. (original) The method of claim 56, wherein the operating system and the thread execute in different execution modes of the computer, and the steps to maintain the association

between the thread and the context are automatically invoked, without explicit software request, on a transition between the thread execution mode and the operating system execution mode.

67. (original) The method of claim 66, wherein the thread execution mode and the operating system execution mode are two different instruction set architectures of the computer.

68. (original) The method of claim 56, wherein the operating system maintains an association between contexts and corresponding threads of execution, each such context including values of data registers, the method further comprising:

modifying at least half of the data registers of the portion of the thread context maintained by the operating system before delivering the thread to the operating system.

69. (original) The method of claim 68, wherein at least some of the modified registers are overwritten by information indicating a storage location at which at least the portion of the thread context to be modified is saved before the modifying.

70. (original) The method of claim 68, wherein at least some of the modified registers are overwritten by a value that enables validation of the contents of the context.

71. (original) The method of claim 56, further comprising the step of modifying a linkage return address for the thread to include information used to restore the context of the thread.

72. (original) The method of claim 71, wherein the linkage register is modified with information indicating an execution path by which, or a condition on which, execution arrived at the entry handler.

73. (original) The method of claim 71, wherein the modification leaves at least half of the bits of the linkage return address intact.

74. (original) The method of claim 71, wherein the linkage register is modified with information indicating a storage location at which at least the portion of the thread context to be modified is saved before the modifying.

75. (previously presented) The method of claim 56:

wherein the interrupted thread at the point of interruption executes in one instruction set architecture and the operating system is coded primarily in a different instruction set architecture; and

further comprising the step of setting of a register to a value that specifies actions to be taken by the entry handler or exit handler to convert operands from one form to another to conform to a data storage convention of the operating system instruction set architecture.

76. (original) The method of claim 56, further comprising either the step of deferring delivery of an interrupt before interrupting the thread by a time sufficient to allow the thread to reach a checkpoint in the execution of the thread where the amount of extended context, being the resources of the thread context beyond those whose resource association with the thread is maintained by the operating system, is reduced.

77. (original) The method of claim 56, further comprising either the step of rolling execution of the thread back to a checkpoint in the execution of the thread where the amount of extended context, being the resources of the thread context beyond those whose resource association with the thread is maintained by the operating system, is reduced.

78. (previously presented) The method of claim 56, further comprising either the step of storing at least the extended context, the extended context being the resources beyond those whose resource association with the thread is maintained by the operating system, into a storage location, a pool of storage locations being managed by a queuing discipline in which empty storage locations in which a context is to be saved are allocated from the head of the queue, recently-emptied storage locations for reuse are enqueued at the head of the queue, and full storage locations to be saved are queued at the tail of the queue.

79. (currently amended) A method, comprising:

during invocation of a service routine of a computer, passing a linkage return address to the service routine at which to resume execution on completion of the service, the linkage return address being deliberately chosen so that an attempt to execute an instruction from the linkage return address on return from the service routine will raise a program execution exception;

on return from the service routine, attempting to execute the instruction at the linkage return address and raising the chosen exception; and

after servicing the exception, placing in execution ~~returning control to~~ a caller of the service routine.

80. (original) The method of claim 79, wherein the passed linkage return address is selected to point to a memory page having a memory attribute that raises the chosen exception on an attempt to execute an instruction from the page.

81. (original) The method of claim 79, wherein

the service routine is an interrupt service routine of an operating system for a computer architecture other than the architecture native to the computer;

the service routine is invoked by an asynchronous interrupt; and

the caller is coded in the instruction set native to the architecture.

82. (previously presented) The method of claim 5, further comprising the step of:

without modifying source code of a pre-existing thread scheduler of the computer, establishing an entry handler for execution at a specified entry point or on a specified entry condition to the thread scheduler, the entry handler programmed to save a context of an interrupted thread and modify the thread context before delivering the modified context to the thread scheduler.

83. (currently amended) The method of claim 20, further comprising the steps of:

during invocation of a service routine of the operating system, passing a linkage return address to the service routine at which to resume execution on completion of the service, the linkage return address being deliberately chosen so that an attempt to execute an instruction from the linkage return address on return from the service routine will raise a program execution exception;

on return from the service routine, attempting to execute the instruction at the linkage return address and raising the chosen exception; and

after servicing the exception, placing in execution ~~returning control to~~ a caller of the service routine.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Serial No.:	09/239,194 Confirmation No.: 9716
Applicant:	John S. Yates, Jr., et al.
Title:	EXECUTING PROGRAMS OF A FIRST COMPUTER ARCHITECTURE ON A COMPUTER OF A SECOND ARCHITECTURE
Filed:	January 28, 1999
Art Unit:	2127
Examiner:	Kenneth Tang
Atty. Docket:	114596-05-4013
Customer No.	38492

I certify that this correspondence, along with any documents referred to therein, is being deposited with the United States Postal Service on March 21, 2005 as First Class Mail in an envelope with sufficient postage addressed to Mail Stop AF, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

AFTER FINAL – EXPEDITED PROCEDURE

REQUEST TO WITHDRAW FINALITY OF OFFICE ACTION

Mail Stop AF
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Applicant observes that the Action of January 13, 2005 was prematurely made final. Pursuant to MPEP § 706.07(c) and (d), Applicant requests that the premature finality of the Action of May 10, 2002 be withdrawn, and that the Response to Office Action filed herewith be entered as of right.

I. The Office Action of May 2002 Raises Three "New Grounds of Rejection," Preventing Finality

An Action may not be made final when it introduces a new ground of rejection, where the new ground was not necessitated by an amendment. MPEP § 706.07(a).

A "new ground of rejection" is any new line of reasoning that requires a "fair opportunity to react to the thrust of the rejection." *In re Kronig*, 539 F.2d 1300, 1302-03, 190 USPQ 425, 426 (CCPA 1976). For example, relying on new portions of the same references, for disclosure not found in portions previously relied on, is a "new ground of rejection." *In re Wiechert*, 370 F.2d, 927, 933, 152 USPQ 247, 251-52 (CCPA 1967) ("An applicant's attention and response are

naturally focused on that portion of the reference which is specifically pointed out by the examiner. ... [W]hen a rejection is factually based on an entirely different portion of an existing reference the appellant should be afforded an opportunity to make a showing of unobviousness vis-à-vis such portion of the reference," emphasis added).

First, at page 5, line 19 and page 20, line 5, the January 2005 Office Action adds col. 33, lines 1-9 of Chernoff '028 as a new ground. (Compare Action of June 2004, page 6, line 17). As discussed in the Accompanying Response to Office Action, § V.B at page 3, this portion of Chernoff discusses entirely different subject matter than the portions of Chernoff '028 previously relied upon. Neither claim 1 nor claim 5 (the two independent claims to which this portion of the Office Action is relevant) were amended in any way relating to this limitation.

Second, in paragraph 102, the Office Action adds a totally new reference, DiBrino 5,371,894, in connection with claims 1 and 79. A new reference, even one offered to back up a previous assertion of official notice, is a new ground of rejection.[14] *In re Ahlert*, 424 F.2d 1088, 1092 n. 4, 165 USPQ 418, 421 n. 4 (CCPA 1970) (when the Board cites a new reference to back up its assertion of official notice, "it is not uncommon for the board itself to cite new references, in which case a new ground of rejection is always stated," emphasis added); *Ex parte Skinkiss*, Appeal No. 2000-0226, http://www.uspto.gov/web/offices/dcom/bpai/decisions/fd000226.pdf at 4 n. 1 (Bd. Pat. App. & Interf.) ("new piece of evidence," even an assertion of "well-known

---

[14] Paragraph 102 states that the addition of DiBrino is not a "new ground of rejection." The Office Action cites no authority that would support this contention.

The Federal Circuit has made clear that no rejection may rely on bald assertion or common sense; rejections may only build on evidence drawn from one of the categories of § 102. *In re Zurko*, 258 F.3d 1379, 1385, 59 USPQ2d 1693, 1697 (Fed. Cir. 2001) ("the Board cannot simply reach conclusions based on its own understanding or experience—or on its assessment of what would be basic knowledge or common sense. Rather, the Board must point to some concrete evidence in the record in support of these findings.") and *In re Lee*, 277 F.3d 1338, 1343-44, 61 USPQ2d 1430, 1434 (Fed. Cir. 2002) ("'common knowledge and common sense' on which the Board relied in rejecting Lee's application are not the specialized knowledge and expertise contemplated by the Administrative Procedure Act. Conclusory statements such as those here provided do not fulfill the agency's obligation... The board cannot rely on conclusory statements when dealing with particular combinations of prior art and specific claims, but must set forth the rationale on which it relies").

If the new DiBrino reference is not part of the ground of rejection, then there is no rejection at all. The Examiner cannot have it both ways.

custom," constitutes "a new ground of rejection"). The claims to which DiBrino might be relevant were not amended in any way that is material to DiBrino.

Third, paragraph 96 for the first time provides some explanation for how "returning control," without the word "the" or "said," can possibly lack "antecedent basis" under § 112 ¶ 2. Without the newly-provided explanation, no applicant could have understood the nature of the rejection. This is a new "thrust" that requires a "fair opportunity to react."

These "new grounds of rejection" render final rejection premature under MPEP § 706.07 and § 706.07(a).

II. The Office Action fails to "Answer All Material Traversed" – an Incomplete Office Action Cannot be Final

MPEP § 707.07(g) warns that "Piecemeal examination should be avoided. as much as possible." MPEP § 706.07 elaborates § 707.07(g), and protects applicants against piecemeal examination, forbidding a rejection from being made final when the rejection is incomplete:

> Before final rejection is in order, a clear issue should be developed between the examiner and applicant. ...

Paragraph 6a of the Office Action of June 2004 raised an issue relating to "extended context" and "modified context." Applicant traversed any rejection, indicating that the basis of the rejection was not understood, and requested further explanation. (Response to Office Action of Sept. 13, 2004, Paragraph IV, at page 19). No such explanation is provided in the Office Action of January 2005, yet the rejection is maintained.

The basis for the rejection is still not understood. As explained in § VI.B.2 of the accompanying Response to Office Action, the most apparent basis for this rejection is directly contrary to instructions in the MPEP.

MPEP § 707.07(f) requires an examiner to "Answer all Material Traversed." The Action of January 2005 is silent on this issue.

The Office Action of January 13, 2005 is incomplete, and therefore may not be made final. If the rejection has any valid legal basis, that explanation must be provided at a time when an applicant has a fair opportunity to respond.

III. Applicant Requests an Interview

The Examiner indicated that no interview would be granted an application under final rejection pursuant to MPEP § 713.09. In the event of grant of this request, Applicant suggests that denial of an interview was premature. In the event that the application is not allowed, Applicant requests an interview. As noted above, Applicant does not understand any of the § 112 ¶ 2 issues – without some explanation that is consonant with MPEP principles, Applicant is unable to respond. Conversely, it appears that most of the § 103 issues arise out of a lack of understanding by the Examiner; Applicant suggests that an interview would be very helpful on these issues as well.

IV. Conclusion

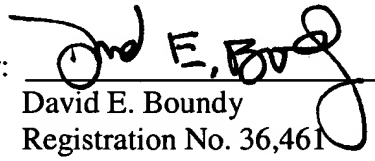
For these reasons, the finality of the Action of January 2005 should be withdrawn, this amendment should be entered as of right (even without a showing of reasons under Rule 116), and these remarks should be given the Examiner's full consideration.

It is believed that this paper occasions no fee. Kindly charge any fee due to Deposit Account No. 23-2405, Order No. 114596-05-4013.

Respectfully submitted,

WILLKIE FARR & GALLAGHER LLP

Dated: March 21, 2005

By: 

David E. Boundy
Registration No. 36,461

WILLKIE FARR & GALLAGHER LLP
787 Seventh Ave.
New York, New York 10019
(212) 728-8000
(212) 728-8111 Fax

Evidence Appendix

Response to Office Action

GA22-7000-7
File No. S370-01

Systems

IBM System/370 Principles of Operation

IBM

Permanently Assigned Real Addresses

- Addresses of PSWs, interruption codes, and associated information used during interruption
- Address used by CPU to update interval timer at real location 80
- Address of CAW, CSW, and other locations used during an I/O interruption or during execution of an I/O instruction, including STORE CHANNEL ID

Absolute Addresses

- Prefix value
- CCW address in CAW
- Data address in CCW
- Address of the indirect-data-address list in a CCW specifying indirect-data addressing
- CCW address in a CCW specifying transfer in channel
- Data address in indirect-data-address words
- IOEL address at real location 172
- Failing-storage address stored in the word at real location 248
- CCW address in CSW

Permanently Assigned Absolute Addresses

- Addresses of PSW and first two CCWs used for initial program loading
- Addresses used for the store-status function

Addresses Not Used to Reference Storage

- PER starting address in control register 10
- PER ending address in control register 11
- The address stored in the word at real location 156 for a monitoring event
- Address in shift instructions and other instructions specified not to use the address to reference storage
- Parameter stored in the word at real location 128 for a service-signal external interruption

Handling of Addresses (Part 2 of 2)

ASSIGNED STORAGE LOCATIONS

ASSIGNED REAL-STORAGE LOCATIONS

The figure "Assigned Locations in Real Storage" shows the format and extent of the assigned locations in real storage. In a multiprocessing system, real storage addresses are transformed to absolute addresses by means of prefixing. The locations are used as follows. Unless specifically noted, the usage applies to both the BC and EC modes.

0-7 Restart New PSW: The new PSW is fetched from locations 0-7 during a restart interruption.

8-15 Restart Old PSW: The current PSW is stored as the old PSW at locations 8-15 during a restart interruption.

24-31 External Old PSW: The current PSW is stored as the old PSW at locations 24-31 during an external interruption.

32-39 Supervisor-Call Old PSW: The current PSW is stored as the old PSW at locations 32-39 during a supervisor-call interruption.

40-47 Program Old PSW: The current PSW is stored as the old PSW at locations 40-47 during a program interruption.

48-55 Machine-Check Old PSW: The current PSW is stored as the old PSW at locations 48-55 during a

machine-check interruption.

56-63 Input/Output Old PSW: The current PSW is stored as the old PSW at locations 56-63 during an I/O interruption.

64-71 CSW: The channel-status word (CSW) is stored at locations 64-71 during an I/O interruption. Part or all of it may be stored during the execution of START I/O, START I/O FAST RELEASE, TEST I/O, CLEAR I/O, HALT I/O, or HALT DEVICE, in which case condition code 1 is set.

72-75 CAW: The channel-address word (CAW) is fetched from locations 72-75 during the execution of START I/O and START I/O FAST RELEASE.

80-83 Interval Timer: Locations 80-83 contain the interval timer. The interval timer is updated whenever the CPU is in the operating state and the manual interval-timer control is set to enable.

84-87 Address of Trace-Table Header: The address of the control block which defines the trace table used by DAS tracing and by the System/370 extended facility is provided in this location.

88-95 External New PSW: The new PSW is fetched from locations 88-95 during an external interruption.

96-103 Supervisor-Call New PSW: The new PSW is fetched from locations 96-103 during a supervisor-call interruption.

104-111 Program New PSW: The new PSW is fetched from locations 104-111 during a program interruption.

112-119 Machine-Check New PSW: The new PSW is fetched from locations 112-119 during a machine-check interruption.

120-127 Input/Output New PSW: The new PSW is fetched from locations 120-127 during an I/O interruption.

128-131 External-Interruption Parameter: During an external interruption due to service signal, the parameter associated with the interruption is stored at locations 128-131.

132-133 CPU Address: During an external interruption due to malfunction alert, emergency signal, or external call, the CPU address associated with the source of the interruption is stored at locations 132-133. For all other external-interruption conditions, zeros are stored at locations 132-133 when the old PSW specified the EC mode, and the field remains unchanged when the old PSW specified the BC mode.

134-135 External-Interruption Code: During an external interruption in the EC mode, the interruption code is stored at locations 134-135.

136-139 Supervisor-Call-Interruption Identification: During a supervisor-call interruption in the EC mode, the instruction-length code is stored in bit positions 5 and 6 of location 137, and the interruption code is stored at locations 138-139. Zeros are stored at location 136 and in the remaining bit positions of 137.

140-143 Program-Interruption Identification: During a program interruption in the EC mode, the instruction-length code is stored in bit positions 5 and 6 of location 141, and the interruption code is stored at locations 142-143. Zeros are stored at location 140 and in the remaining bit positions of 141.

144-147 Translation-Exception Identification: During a program interruption due to a segment-translation exception or a page-translation exception, the virtual address being translated is stored at locations 144-147. This address is sometimes referred to as the translation-exception address. Bits 1-7 of location 144 are set to zeros. With DAS, bit 0 of location 144 is set to zero if the translation was relative to the primary segment table designated by control register 1, and set to one if the translation was relative to the secondary segment table designed by control register 7. Without DAS, bit 0 of location 144 is set to zero.

During a program interruption due to an AFX-translation, ASX-translation, primary-authority, or secondary-authority exception, the ASN being translated is stored at locations 146-147. Locations 144-145 are set to zeros.

During a program interruption for a space-switching event, the old PASN, which appears in the right half of control register 4 before the execution of a space-switching

PC or PT instruction, is stored at locations 146-147. Locations 144-145 are set to zeros.

During a program interruption due to an LX-translation or EX-translation exception, the PC number is stored in bit positions 12-31 of the word at location 144. Bits 0-11 are set to zeros.

148-149 Monitor-Class Number: During a program interruption due to a monitor event, the monitor-class number is stored at location 149, and zeros are stored at 148.

150-151 PER Code: During a program interruption due to a PER event, the PER code is stored in bit positions 0-3 of location 150, and zeros are stored in bit positions 4-7 and at location 151. This field can be stored only when the instruction causing the PER condition was executed under the control of a PSW specifying the EC mode.

152-155 PER Address: During a program interruption due to a program event, the PER address is stored at locations 153-155, and zeros are stored at location 152. This field can be stored only when the instruction causing the PER condition was executed under the control of a PSW specifying the EC mode.

156-159 Monitor Code: During a program interruption due to a monitor event, the monitor code is stored at locations 157-159, and zeros are stored at location 156.

161-163 MAPL: Address of a control block used by the System/370 extended facility.

168-171 Channel ID: The four-byte channel-identification information is stored at locations 168-171 during the execution of STORE CHANNEL ID.

172-175 IOEL Address: The I/O-extended-logout address is fetched from locations 172-175 during the I/O-extended-logout operation.

176-179 Limited Channel Logout: The limited-channel-logout information is stored at locations 176-179. This field may be stored only when the CSW or a portion of the CSW is stored.

185-187 I/O Address: During an I/O interruption in the EC mode, the two-byte I/O address is stored at locations 186-187, and zeros are stored at location 185.

216-223 Machine-Check CPU-Timer Save Area: During a machine-check interruption, the contents of the CPU timer, if installed, are stored at locations 216-223.

224-231 Machine-Check Clock-Comparator Save Area: During a machine-check interruption, the contents of the clock comparator, if installed, are stored at location 224-231.

232-239 Machine-Check-Interruption Code: During a machine-check interruption the machine-check-interruption code is stored at locations 232-239.

244-247 External-Damage Code: During a machine-check interruption due to certain external-damage conditions, depending on the model, an external-damage code may be stored in these locations.

248-251 Failing-Storage Address: During a machine-check interruption, a failing-storage address, if any, is stored at locations 249-251, and zeros are stored at location 248.

252-255 Region Code: During a machine-check interruption, model-dependent information may be stored at locations 252-255.

256-351 Fixed-Logout Area: Depending on the model, logout information may be placed in this area during a machine-check interruption. Additionally, the contents of locations 256-351 may be changed at any time, subject to the asynchronous-fixed-logout-control bit in control register 14.

352-383 Machine-Check Floating-Point-Register Save Area: During a machine-check interruption, the contents of the floating-point registers are stored at locations 352-383.

384-447 Machine-Check General-Register Save Area: During a machine-check interruption, the contents of the general registers are stored at locations 384-447.

448-511 Machine-Check Control-Register Save Area: During a machine-check interruption, the contents of the control registers are stored at locations 448-511.

ASSIGNED ABSOLUTE STORAGE LOCATIONS

The figure "Assigned Locations in Absolute Storage" shows the format and extent of the assigned locations in absolute storage. The locations are as follows, and the usage applies to both the BC and EC modes.

0-7 IPL PSW: The first eight bytes read during the IPL initial read operation are stored at locations 0-7. The contents of these locations are used as the new PSW at the completion of the IPL operation. These locations may also be used for temporary storage at the initiation of the IPL operation.

8-15 IPL CCW1: Bytes 8-15 read during the IPL initial read operation are stored at locations 8-15. The contents of these locations are ordinarily used as the next CCW in an IPL CCW chain after completion of the IPL initial-read operation.

16-23 IPL CCW2: Bytes 16-23 read during the IPL initial read operation are stored at locations 16-23. The contents of these locations may be used as another CCW in the IPL CCW chain to follow IPL CCW1.

216-223 Store-Status CPU-Timer Save Area: During the execution of the store-status operation, the contents of the CPU timer, if installed, are stored at locations 216-223.

224-231 Store-Status Clock-Comparator Save Area: During the execution of the store-status operation, the contents of the clock comparator, if installed, are stored at location 224-231.

256-263 Store-Status PSW Save Area: During the execution of the store-status operation, the contents of the current PSW are stored at location 256-263.

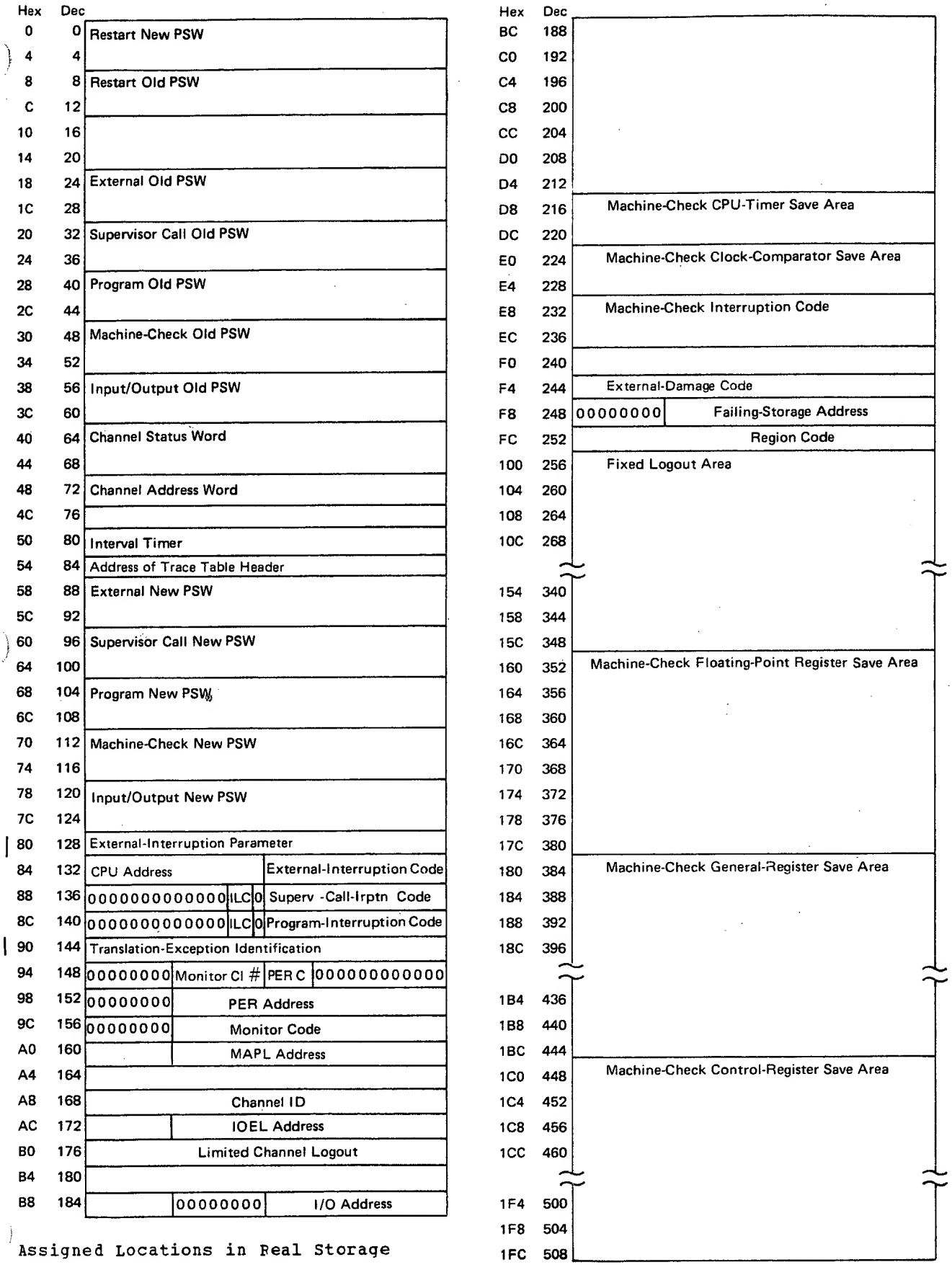
264-267 Store-Status Prefix Save Area: During the execution of the store-status operation, the contents of the prefix register, if installed, are stored at location 264-267.

268-271 Store-Status Model-Dependent Save Area: During the execution of the store-status operation, model-dependent information may be stored at locations 268-271.

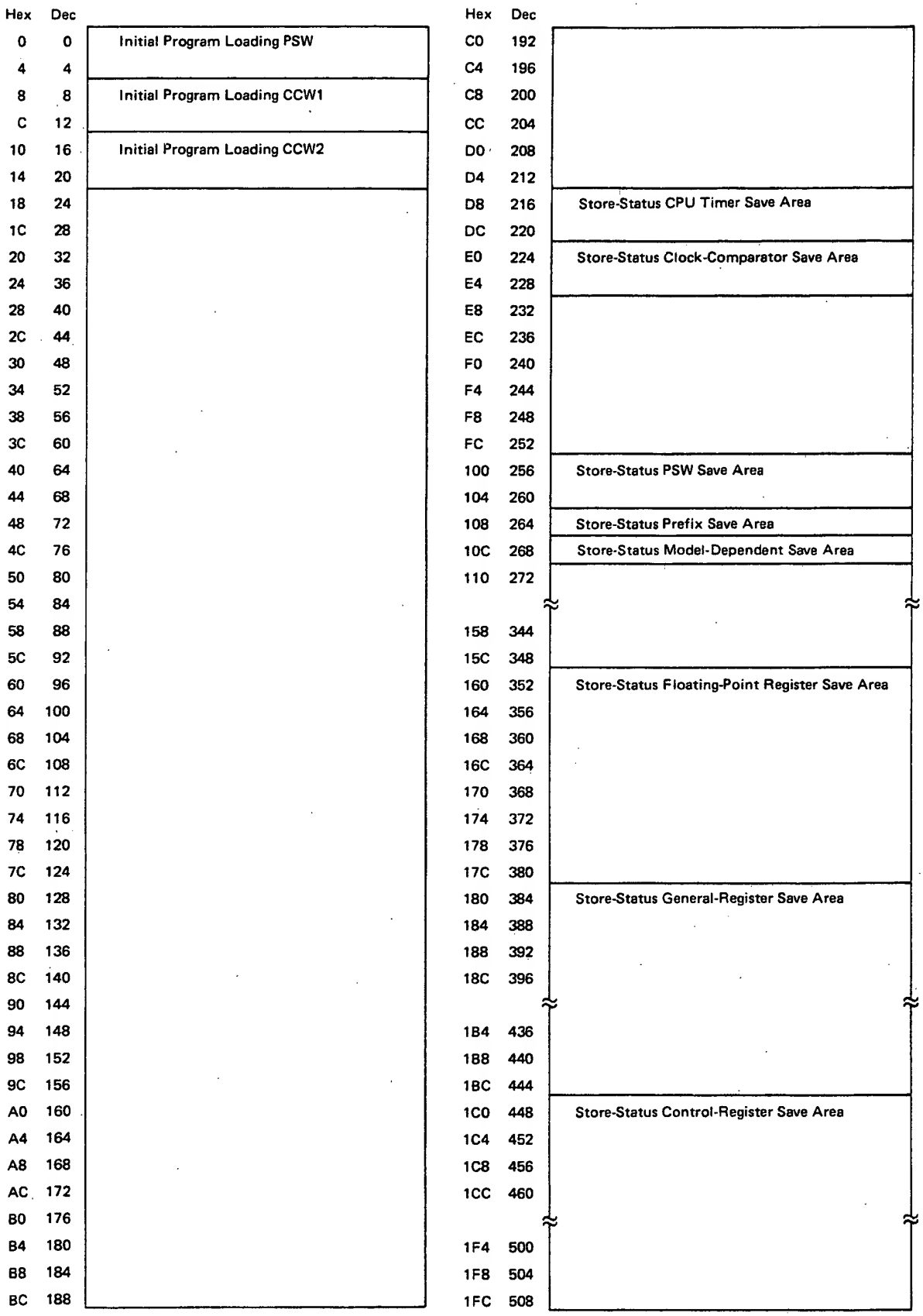
352-383 Store-Status Floating-Point-Register Save Area: During the execution of the store-status operation, the contents of the floating-point registers are stored at locations 352-383.

384-447 Store-Status General-Register Save Area: During the execution of the store-status operation, the contents of the general registers are stored at locations 384-447.

448-511 Store-Status Control-Register Save Area: During the execution of the store-status operation, the contents of the control registers are stored at locations 448-511.



Assigned Locations in Real Storage



Assigned Locations in Absolute Storage

intel®

Intel Architecture Software Developer's Manual

Volume 3: System Programming Guide

NOTE: The *Intel Architecture Developer's Manual* consists of three books: *Basic Architecture*, Order Number 243190; *Instruction Set Reference Manual*, Order Number 243191; and the *System Programming Guide*, Order Number 243192. Please refer to all three volumes when evaluating your design needs.

1997

■

intel®

CHAPTER 6
TASK MANAGEMENT

This chapter describes the Intel Architecture's task management facilities. These facilities are only available when the processor is running in protected mode.

6.1. TASK MANAGEMENT OVERVIEW

A task is a unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility.

The Intel Architecture provides a mechanism for saving the state of a task, for dispatching tasks for execution, and for switching from one task to another. When operating in protected mode, all processor execution takes place from within a task. Even simple systems must define at least one task. More complex systems can use the processor's task management facilities to support multitasking applications.

6.1.1. Task Structure

A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment, and one or more data segments (see Figure 6-1). If an operating system or executive uses the processor's privilege-level protection mechanism, the task execution space also provides a separate stack for each privilege level.

The TSS specifies the segments that make up the task execution space and provides a storage place for task state information. In multitasking systems, the TSS also provides a mechanism for linking tasks.

NOTE

This chapter describes primarily 32-bit tasks and the 32-bit TSS structure. For information on 16-bit tasks and the 16-bit TSS structure, see Section 6.6., "16-Bit Task-State Segment (TSS)".

A task is identified by the segment selector for its TSS. When a task is loaded into the processor for execution, the segment selector, base address, limit, and segment descriptor attributes for the TSS are loaded into the task register (see Section 2.4.4., "Task Register (TR)").

If paging is implemented for the task, the base address of the page directory used by the task is loaded into control register CR3.

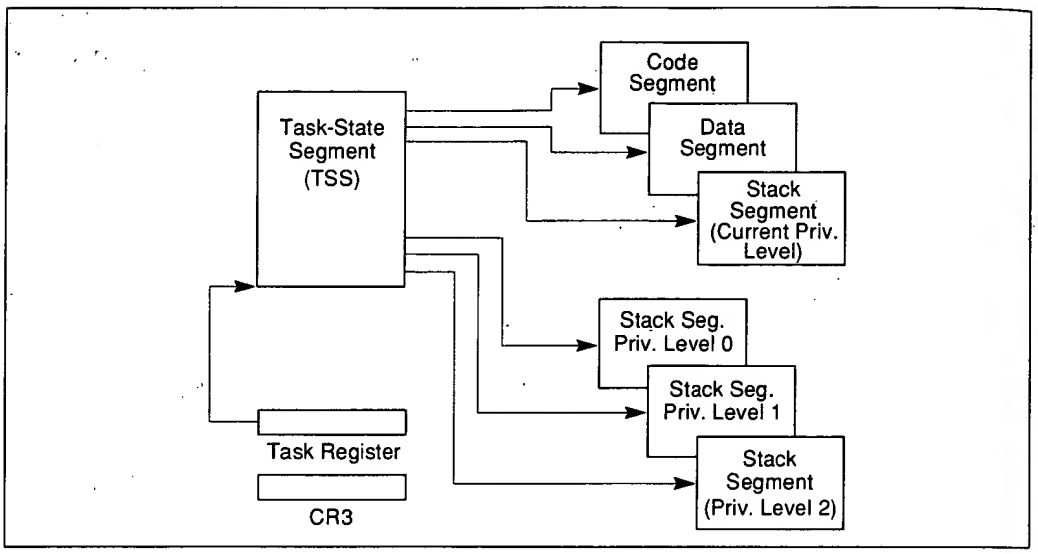


Figure 6-1. Structure of a Task

6.1.2. Task State

The following items define the state of the currently executing task:

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).
- The state of the general-purpose registers.
- The state of the EFLAGS register.
- The state of the EIP register.
- The state of control register CR3.
- The state of the task register.
- The state of the LDTR register.
- The I/O map base address and I/O map (contained in the TSS).
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS).
- Link to previously executed task (contained in the TSS).

Prior to dispatching a task, all of these items are contained in the task's TSS, except the state of the task register. Also, the complete contents of the LDTR register are not contained in the TSS, only the segment selector for the LDT.

6.1.3. Executing a Task

Software or the processor can dispatch a task for execution in one of the following ways:

- A explicit call to a task with the CALL instruction.
- A explicit jump to a task with the JMP instruction.
- An implicit call (by the processor) to an interrupt-handler task.
- An implicit call to an exception-handler task.
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods of dispatching a task identify the task to be dispatched with a segment selector that points either to a task gate or the TSS for the task. When dispatching a task with a CALL or JMP instruction, the selector in the instruction may select either the TSS directly or a task gate that holds the selector for the TSS. When dispatching a task to handle an interrupt or exception, the IDT entry for the interrupt or exception must contain a task gate that holds the selector for the interrupt- or exception-handler TSS.

When a task is dispatched for execution, a task switch automatically occurs between the currently running task and the dispatched task. During a task switch, the execution environment of the currently executing task (called the task's state or **context**) is saved in its TSS and execution of the task is suspended. The context for the dispatched task is then loaded into the processor and execution of that task begins with the instruction pointed to by the newly loaded EIP register. If the task has not been run since the system was last initialized, the EIP will point to the first instruction of the task's code; otherwise, it will point to the next instruction after the last instruction that the task executed when it was last active.

If the currently executing task (the calling task) called the task being dispatched (the called task), the TSS segment selector for the calling task is stored in the TSS of the called task to provide a link back to the calling task.

For all Intel Architecture processors, tasks are not recursive. A task cannot call or jump to itself.

Interrupts and exceptions can be handled with a task switch to a handler task. Here, the processor not only can perform a task switch to handle the interrupt or exception, but it can automatically switch back to the interrupted task upon returning from the interrupt- or exception-handler task. This mechanism can handle interrupts that occur during interrupt tasks.

As part of a task switch, the processor can also switch to another LDT, allowing each task to have a different logical-to-physical address mapping for LDT-based segments. The page-directory base register (CR3) also is reloaded on a task switch, allowing each task to have its own set of page tables. These protection facilities help isolate tasks and prevent them from interfering with one another. If one or both of these protection mechanisms are not used, the processor provides no protection between tasks. This is true even with operating systems that use multiple privilege levels for protection. Here, a task running at privilege level 3 that uses the same LDT and page tables as other privilege-level-3 tasks can access code and corrupt data and the stack of other tasks.

Use of task management facilities for handling multitasking applications is optional. Multitasking can be handled in software, with each software defined task executed in the context of a single Intel Architecture task.

6.2. TASK MANAGEMENT DATA STRUCTURES

The processor defines five data structures for handling task-related activities:

- Task-state segment (TSS).
- Task-gate descriptor.
- TSS descriptor.
- Task register.
- NT flag in the EFLAGS register.

When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the segment selector for the TSS must be loaded into the task register (using the LTR instruction).

6.2.1. Task-State Segment (TSS)

The processor state information needed to restore a task is saved in a system segment called the task-state segment (TSS). Figure 6-2 shows the format of a TSS for tasks designed for 32-bit CPUs. (Compatibility with 16-bit Intel 286 processor tasks is provided by a different kind of TSS, see Figure 6-9.) The fields of a TSS are divided into two main categories: dynamic fields and static fields.

The processor updates the dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

General-purpose register fields
: State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.

Segment selector fields
: Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.

EFLAGS register field
: State of the EFAGS register prior to the task switch.

EIP (instruction pointer) field
: State of the EIP register prior to the task switch.

Previous task link field
: Contains the segment selector for the TSS of the previous task (updated on a task switch that was initiated by a call, interrupt, or exception). This field

(which is sometimes called the back link field) permits a task switch back to the previous task to be initiated with an IRET instruction.

processor reads the static fields, but does not normally change them. These fields are set up a task is created. The following are static fields:

segment selector field
Contains the segment selector for the task's LDT.

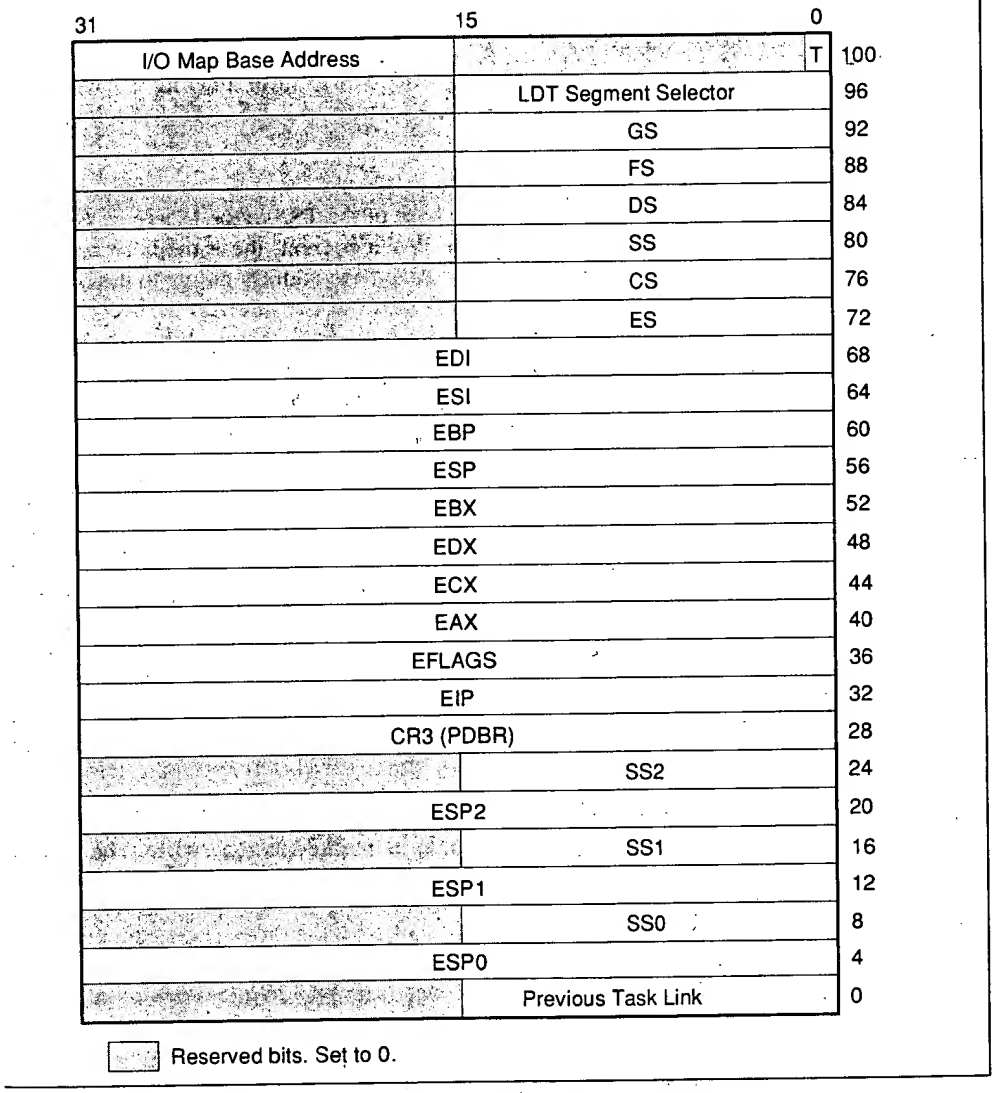


Figure 6-2. 32-Bit Task-State Segment (TSS)

CR3 control register field

Contains the base physical address of the page directory to be used by the task. Control register CR3 is also known as the page-directory base register (PDBR).

Privilege level-0, -1, and -2 stack pointer fields

These stack pointers consist of a logical address made up of the segment selector for the stack segment (SS0, SS1, and SS2) and an offset into the stack (ESP0, ESP1, and ESP2). Note that the values in these fields are static for a particular task; whereas, the SS and ESP values will change if stack switching occurs within the task.

T (debug trap) flag (byte 100, bit 0)

When set, the T flag causes the processor to raise a debug exception when a task switch to this task occurs (see Section 14.3.1.5., "Task-Switch Exception Condition").

I/O map base address field

Contains a 16-bit offset from the base of the TSS to the I/O permission bit map and interrupt redirection bitmap. When present, these maps are stored in the TSS at higher addresses. The I/O map base address points to the beginning of the I/O permission bit map and the end of the interrupt redirection bit map. See Chapter 9, *Input/Output*, in the *Intel Architecture Software Developer's Manual, Volume 1*, for more information about the I/O permission bit map. See Section 15.3., "Interrupt and Exception Handling in Virtual-8086 Mode", for a detailed description of the interrupt redirection bit map.

If paging is used, care should be taken to avoid placing a page boundary within the part of the TSS that the processor reads during a task switch (the first 104 bytes). If a page boundary is placed within this part of the TSS, the pages on either side of the boundary must be present at the same time and contiguous in physical memory. The reason for this restriction is that when accessing a TSS during a task switch, the processor reads and writes into the first 104 bytes of each TSS from contiguous physical addresses beginning with the physical address of the first byte of the TSS. It may not perform address translations at a page boundary if one occurs within this area. So, after the TSS access begins, if a part of the 104 bytes is not both present and physically contiguous, the processor will access incorrect TSS information, without generating a page-fault exception. The reading of this incorrect information will generally lead to an unrecoverable exception later in the task switch process.

Also, if paging is used, the pages corresponding to the previous task's TSS, the current task's TSS, and the descriptor table entries for each should be marked as read/write. The task switch will be carried out faster if the pages containing these structures are also present in memory before the task switch is initiated.

6.2.2. TSS Descriptor

The TSS, like all other segments, is defined by a segment descriptor. Figure 6-3 shows the format of a TSS descriptor. TSS descriptors may only be placed in the GDT; they cannot be placed in an LDT or the IDT. An attempt to access a TSS using a segment selector with its TI flag set (which indicates the current LDT) causes a general-protection exception (#GP) to be

generated. A general-protection exception is also generated if an attempt is made to load a segment selector for a TSS into a segment register.

The busy flag (B) in the type field indicates whether the task is busy. A busy task is currently running or is suspended. A type field with a value of 1001B indicates an inactive task; a value of 1011B indicates a busy task. Tasks are not recursive. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To insure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.

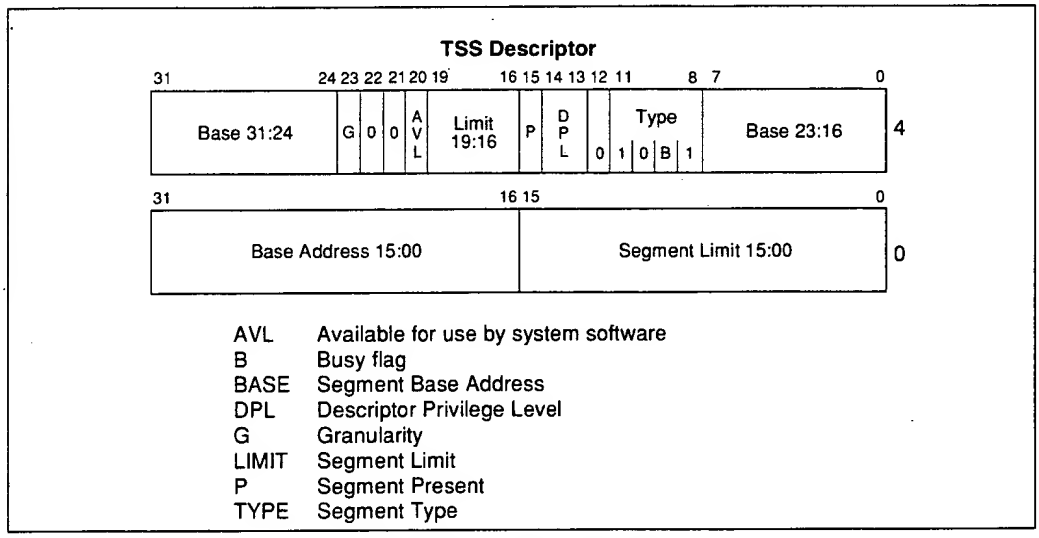


Figure 6-3. TSS Descriptor

The base, limit, and DPL fields and the granularity and present flags have functions similar to their use in data-segment descriptors (see Section 3.4.3., "Segment Descriptors"). The limit field must have a value equal to or greater than 67H (for a 32-bit TSS), one byte less than the minimum size of a TSS. Attempting to switch to a task whose TSS descriptor has a limit less than 67H generates an invalid-TSS exception (#TS). A larger limit is required if an I/O permission bit map is included in the TSS. An even larger limit would be required if the operating system stores additional data in the TSS. The processor does not check for a limit greater than 67H on a task switch; however, it does when accessing the I/O permission bit map or interrupt redirection bit map.

Any program or procedure with access to a TSS descriptor (that is, whose CPL is numerically equal to or less than the DPL of the TSS descriptor) can dispatch the task with a call or a jump. In most systems, the DPLs of TSS descriptors should be set to values less than 3, so that only privileged software can perform task switching. However, in multitasking applications, DPLs for some TSS descriptors can be set to 3 to allow task switching at the application (or user) privilege level.

6.2.3. Task Register

The task register holds the 16-bit segment selector and the entire segment descriptor (32-bit base address, 16-bit segment limit, and descriptor attributes) for the TSS of the current task (see Figure 2-4). This information is copied from the TSS descriptor in the GDT for the current task. Figure 6-4 shows the path the processor uses to accesses the TSS, using the information in the task register.

The task register has both a visible part (that can be read and changed by software) and an invisible part (that is maintained by the processor and is inaccessible by software). The segment selector in the visible portion points to a TSS descriptor in the GDT. The processor uses the invisible portion of the task register to cache the segment descriptor for the TSS. Caching these values in a register makes execution of the task more efficient, because the processor does not need to fetch these values from memory to reference the TSS of the current task.

The LTR (load task register) and STR (store task register) instructions load and read the visible portion of the task register. The LTR instruction loads a segment selector (source operand) into the task register that points to a TSS descriptor in the GDT, and then loads the invisible portion of the task register with information from the TSS descriptor. This instruction is a privileged instruction that may be executed only when the CPL is 0. The LTR instruction generally is used during system initialization to put an initial value in the task register. Afterwards, the contents of the task register are changed implicitly when a task switch occurs.

The STR (store task register) instruction stores the visible portion of the task register in a general-purpose register or memory. This instruction can be executed by code running at any privilege level, to identify the currently running task; however, it is normally used only by operating system software.

On power up or reset of the processor, the segment selector and base address are set to the default value of 0 and the limit is set to FFFFH.

6.2.4. Task-Gate Descriptor

A task-gate descriptor provides an indirect, protected reference to a task. Figure 6-5 shows the format of a task-gate descriptor. A task-gate descriptor can be placed in the GDT, an LDT, or the IDT.

The TSS segment selector field in a task-gate descriptor points to a TSS descriptor in the GDT. The RPL in this segment selector is not used.

The DPL of a task-gate descriptor controls access to the TSS descriptor during a task switch. When a program or procedure makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. (Note that when a task gate is used, the DPL of the destination TSS descriptor is not used.)

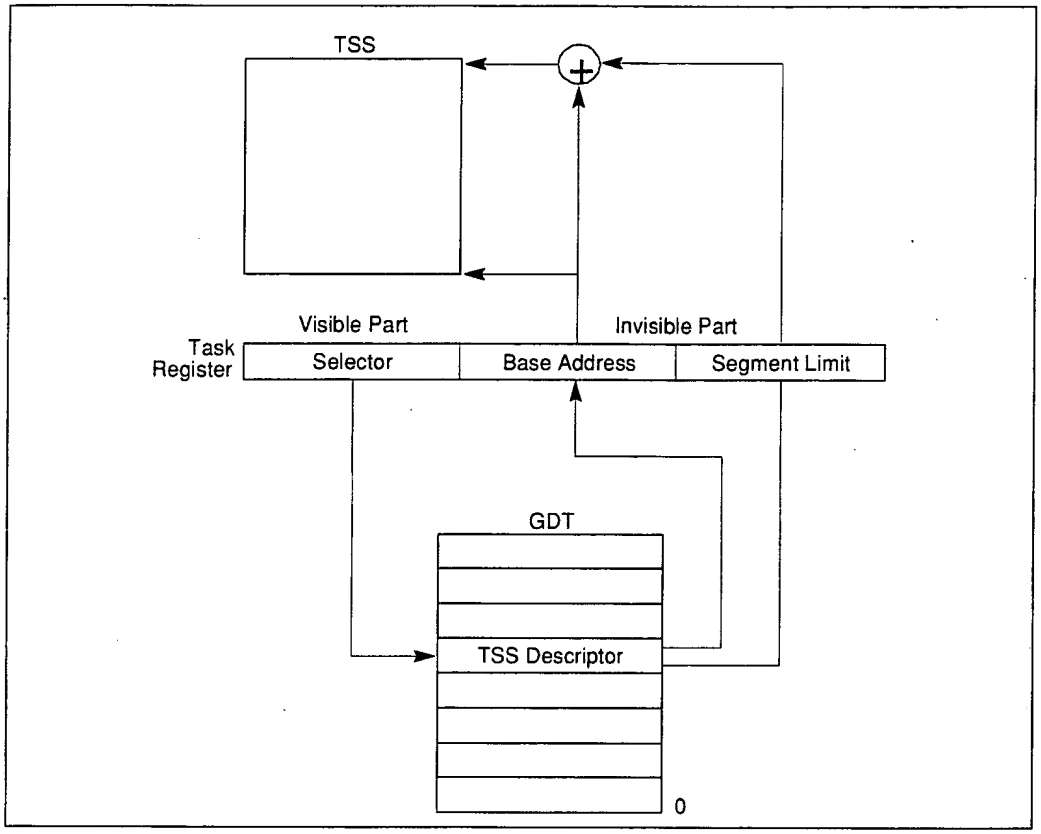


Figure 6-4. Task Register

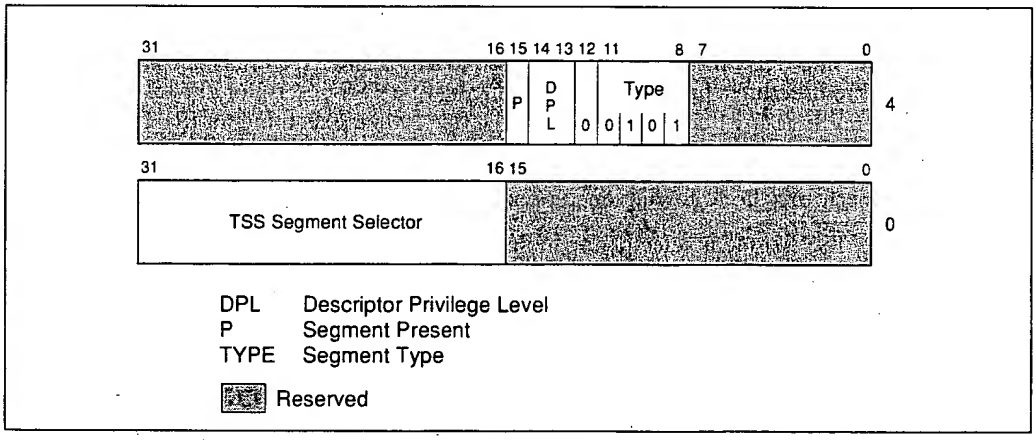


Figure 6-5. Task-Gate Descriptor

A task can be accessed either through a task-gate descriptor or a TSS descriptor. Both of these structures are provided to satisfy the following needs:

- The need for a task to have only one busy flag. Because the busy flag for a task is stored in the TSS descriptor, each task should have only one TSS descriptor. There may, however, be several task gates that reference the same TSS descriptor.
- The need to provide selective access to tasks. Task gates fill this need, because they can reside in an LDT and can have a DPL that is different from the TSS descriptor's DPL. A program or procedure that does not have sufficient privilege to access the TSS descriptor for a task in the GDT (which usually has a DPL of 0) may be allowed access to the task through a task gate with a higher DPL. Task gates give the operating system greater latitude for limiting access to specific tasks.
- The need for an interrupt or exception to be handled by an independent task. Task gates may also reside in the IDT, which allows interrupts and exceptions to be handled by handler tasks. When an interrupt or exception vector points to a task gate, the processor switches to the specified task.

Figure 6-6 illustrates how a task gate in an LDT, a task gate in the GDT, and a task gate in the IDT can all point to the same task.

6.3. TASK SWITCHING

The processor transfers execution to another task in any of four cases:

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.
- The current task executes an IRET when the NT flag in the EFLAGS register is set.

The JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all generalized mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task:

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).

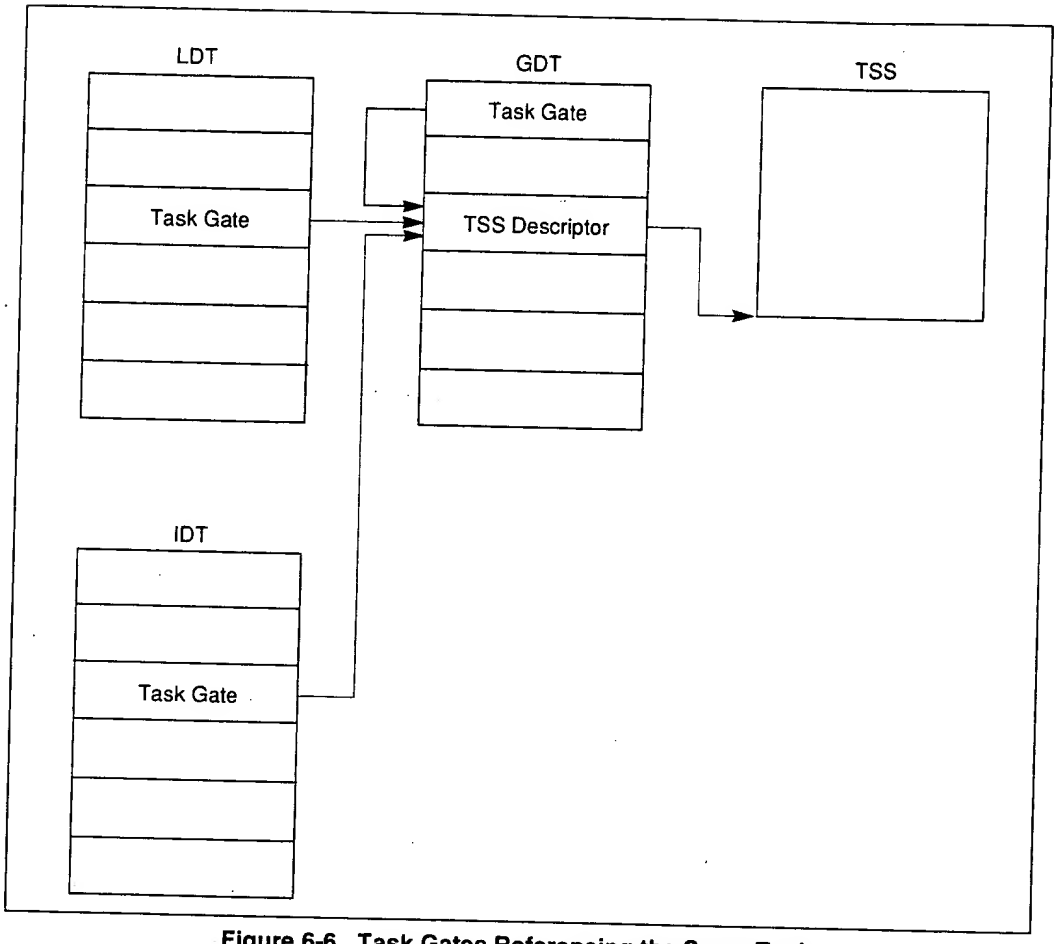


Figure 6-6. Task Gates Referencing the Same Task

2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for interrupts generated by the INT *n* instruction), and the IRET instruction are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT *n* instruction, the DPL is checked.
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H).
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).

5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task's TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt, the busy (B) flag is left set. (See Table 6-2.)
7. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.
8. Saves the state of the current (old) task in the current task's TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).

NOTE

At this point, if all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 8, the processor does not complete the task switch and insures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch. If an unrecoverable error occurs after the commit point (in steps 9 through 14), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task. If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the task. See Chapter 5, "Interrupt 10—Invalid TSS Exception (#TS)", for more information about the affect of exceptions on a task when they occur after the commit point of a task switch.

9. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor sets the NT flag in the EFLAGS image stored in the new task's TSS; if initiated with an IRET instruction, the processor restores the NT flag from the EFLAGS image stored on the stack. If initiated with a JMP instruction, the NT flag is left unchanged. (See Table 6-2.)
10. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.
11. Sets the TS flag in the control register CR0 image stored in the new task's TSS.
12. Loads the task register with the segment selector and descriptor for the new task's TSS.

13. Loads the new task's state from its TSS into processor. Any errors associated with the loading and qualification of segment descriptors in this step occur in the context of the new task. The task state information that is loaded here includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment descriptor parts of the segment registers.
14. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

The state of the currently executing task is always saved when a successful task switch occurs. If the task is resumed, execution starts with the instruction pointed to by the saved EIP value, and the registers are restored to the values they held when the task was suspended.

When switching tasks, the privilege level of the new task does not inherit its privilege level from the suspended task. The new task begins executing at the privilege level specified in the CPL field of the CS register, which is loaded from the TSS. Because tasks are isolated by their separate address spaces and TSSs and because privilege rules control access to a TSS, software does not need to perform explicit privilege checks on a task switch.

Table 6-1 shows the exception conditions that the processor checks for when switching tasks. It also shows the exception that is generated for each check if an error is detected and the segment that the error code references. (The order of the checks in the table is the order used in the P6 family processors. The exact order is model specific and may be different for other Intel Architecture processors.) Exception handlers designed to handle these exceptions may be subject to recursive calls if they attempt to reload the segment selector that generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

Table 6-1. Exception Conditions Checked During a Task Switch

Condition Checked	Exception[1]	Error Code Reference[2]
Segment selector for a TSS descriptor references the GDT and is within the limits of the table.	#GP	New Task's TSS
TSS descriptor is present in memory.	#NP	New Task's TSS
TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).	#GP (for JMP, CALL, INT)	Task's back-link TSS
TSS descriptor is not busy (for task switch initiated by an IRET instruction).	#TS (for IRET)	New Task's TSS
TSS segment limit greater than or equal to 108 (for 32-bit TSS) or 44 (for 16-bit TSS).	#TS	New Task's TSS
Registers are loaded from the values in the TSS.		
LDT segment selector of new task is valid [3].	#TS	New Task's LDT
Code segment DPL matches segment selector RPL.	#TS	New Code Segment
SS segment selector is valid [2].	#TS	New Stack Segment
Stack segment is present in memory.	#SF	New Stack Segment

Table 6-1. Exception Conditions Checked During a Task Switch (Contd.)

Stack segment DPL matches CPL.	#TS	New stack segment
LDT of new task is present in memory.	#TS	New Task's LDT
CS segment selector is valid [3].	#TS	New Code Segment
Code segment is present in memory.	#NP	New Code Segment
Stack segment DPL matches selector RPL.	#TS	New Stack Segment
DS, ES, FS, and GS segment selectors are valid [3].	#TS	New Data Segment
DS, ES, FS, and GS segments are readable.	#TS	New Data Segment
DS, ES, FS, and GS segments are present in memory.	#NP	New Data Segment
DS, ES, FS, and GS segment DPL greater than or equal to CPL (unless these are conforming segments).	#TS	New Data Segment

NOTES:

1. #NP is segment-not-present exception, #GP is general-protection exception, #TS is invalid-TSS exception, and #SF is stack-fault exception.
2. The error code contains an index to the segment descriptor referenced in this column.
3. A segment selector is valid if it is in a compatible type of table (GDT or LDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (for example, a segment selector in the CS register only is valid when it points to a code-segment descriptor).

The TS (task switched) flag in the control register CR0 is set every time a task switch occurs. System software uses the TS flag to coordinate the actions of floating-point unit when generating floating-point exceptions with the rest of the processor. The TS flag indicates that the context of the floating-point unit may be different from that of the current task. See Section 2.5., "Control Registers", for a detailed description of the function and use of the TS flag.

6.4. TASK LINKING

The previous task link field of the TSS (sometimes called the "backlink") and the NT flag in the EFLAGS register are used to return execution to the previous task. The NT flag indicates whether the currently executing task is nested within the execution of another task, and the previous task link field of the current task's TSS holds the TSS selector for the higher-level task in the nesting hierarchy, if there is one (see Figure 6-7).

When a CALL instruction, an interrupt, or an exception causes a task switch, the processor copies the segment selector for the current TSS into the previous task link field of the TSS for the new task, and then sets the NT flag in the EFLAGS register. The NT flag indicates that the previous task link field of the TSS has been loaded with a saved TSS segment selector. If software uses an IRET instruction to suspend the new task, the processor uses the value in the previous task link field and the NT flag to return to the previous task; that is, if the NT flag is set, the processor performs a task switch to the task specified in the previous task link field.

NOTE

When a JMP instruction causes a task switch, the new task is not nested; that is, the NT flag is set to 0 and the previous task link field is not used. A JMP instruction is used to dispatch a new task when nesting is not desired.

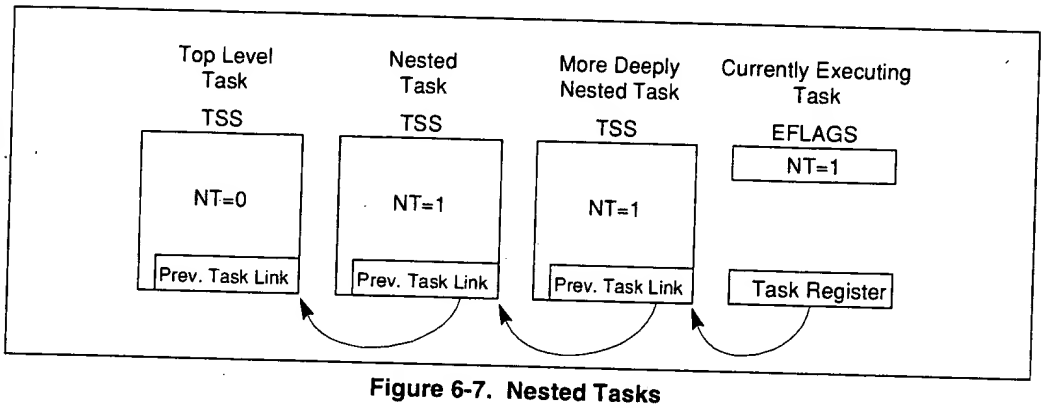


Figure 6-7. Nested Tasks

Table 6-2 summarizes the uses of the busy flag (in the TSS segment descriptor), the NT flag, the previous task link field, and TS flag (in control register CR0) during a task switch. Note that the NT flag may be modified by software executing at any privilege level. It is possible for a program to set its NT flag and execute an IRET instruction, which would have the effect of invoking the task specified in the previous link field of the current task's TSS. To keep spurious task switches from succeeding, the operating system should initialize the previous task link field for every TSS it creates to 0.

Table 6-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag

Flag or Field	Effect of JMP instruction	Effect of CALL Instruction or Interrupt	Effect of IRET Instruction
Busy (B) flag of new task.	Flag is set. Must have been clear before.	Flag is set. Must have been clear before.	No change. Must have been set.
Busy flag of old task.	Flag is cleared.	No change. Flag is currently set.	Flag is cleared.
NT flag of new task.	No change.	Flag is set.	Restored to value from TSS of new task.
NT flag of old task.	No change.	No change.	Flag is cleared.
Previous task link field of new task.	No change.	Loaded with selector for old task's TSS.	No change.
Previous task link field of old task.	No change.	No change.	No change.
TS flag in control register CR0.	Flag is set.	Flag is set.	Flag is set.

6.4.1. Use of Busy Flag To Prevent Recursive Task Switching

A TSS allows only one context to be saved for a task; therefore, once a task is called (dispatched), a recursive (or re-entrant) call to the task would cause the current state of the task to be lost. The busy flag in the TSS segment descriptor is provided to prevent re-entrant task switching and subsequent loss of task state information. The processor manages the busy flag as follows:

1. When dispatching a task, the processor sets the busy flag of the new task.
2. If during a task switch, the current task is placed in a nested chain (the task switch is being generated by a CALL instruction, an interrupt, or an exception), the busy flag for the current task remains set.
3. When switching to the new task (initiated by a CALL instruction, interrupt, or exception), the processor generates a general-protection exception (#GP) if the busy flag of the new task is already set. (If the task switch is initiated with an IRET instruction, the exception is not raised because the processor expects the busy flag to be set.)
4. When a task is terminated by a jump to a new task (initiated with a JMP instruction in the task code) or by an IRET instruction in the task code, the processor clears the busy flag, returning the task to the "not busy" state.

In this manner the processor prevents recursive task switching by preventing a task from switching to itself or to any task in a nested chain of tasks. The chain of nested suspended tasks may grow to any length, due to multiple calls, interrupts, or exceptions. The busy flag prevents a task from being invoked if it is in this chain.

The busy flag may be used in multiprocessor configurations, because the processor follows a LOCK protocol (on the bus or in the cache) when it sets or clears the busy flag. This lock keeps two processors from invoking the same task at the same time. (See Section 7.1.2.1., "Automatic Locking", for more information about setting the busy flag in a multiprocessor applications.)

6.4.2. Modifying Task Linkages

In a uniprocessor system, in situations where it is necessary to remove a task from a chain of linked tasks, use the following procedure to remove the task:

1. Disable interrupts.
2. Change the previous task link field in the TSS of the pre-empting task (that is, the task that suspended the task to be removed). It is assumed that the pre-empting task is the next task (newer task) in the chain from the task to be removed. The previous task link field should be changed to point to the TSS of the next oldest task in the chain or to an even older task in the chain.
3. Clear the busy (B) flag in the TSS segment descriptor for the task being removed from the chain. If more than one task is being removed from the chain, the busy flag for each task being remove must be cleared.
4. Enable interrupts.

In a multiprocessing system, additional synchronization and serialization operations must be added to this procedure to insure that the TSS and its segment descriptor are both locked when the previous task link field is changed and the busy flag is cleared.

6.5. TASK ADDRESS SPACE

The address space for a task consists of the segments that the task can access. These segments include the code, data, stack, and system segments referenced in the TSS and any other segments accessed by the task code. These segments are mapped into the processor's linear address space, which is in turn mapped into the processor's physical address space (either directly or through paging).

The LDT segment field in the TSS can be used to give each task its own LDT. Giving a task its own LDT allows the task address space to be isolated from other tasks by placing the segment descriptors for all the segments associated with the task in the task's LDT.

It also is possible for several tasks to use the same LDT. This is a simple and memory-efficient way to allow some tasks to communicate with or control each other, without dropping the protection barriers for the entire system.

Because all tasks have access to the GDT, it also is possible to create shared segments accessed through segment descriptors in this table.

If paging is enabled, the CR3 register (PDBR) field in the TSS allows each task can also have its own set of page tables for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

6.5.1. Mapping Tasks to the Linear and Physical Address Spaces

Tasks can be mapped to the linear address space and physical address space in either of two ways:

- One linear-to-physical address space mapping is shared among all tasks. When paging is not enabled, this is the only choice. Without paging, all linear addresses map to the same physical addresses. When paging is enabled, this form of linear-to-physical address space mapping is obtained by using one page directory for all tasks. The linear address space may exceed the available physical space if demand-paged virtual memory is supported.
- Each task has its own linear address space that is mapped to the physical address space. This form of mapping is accomplished by using a different page directory for each task. Because the PDBR (control register CR3) is loaded on each task switch, each task may have a different page directory.

The linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses.

With either method of mapping task linear address spaces, the TSSs for all tasks must lie in a shared area of the physical space, which is accessible to all tasks. This mapping is required so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear address space mapped by the GDT also should be mapped to a shared area of the physical space; otherwise, the purpose of the GDT is defeated. Figure 6-8 shows how the linear address spaces of two tasks can overlap in the physical space by sharing page tables.

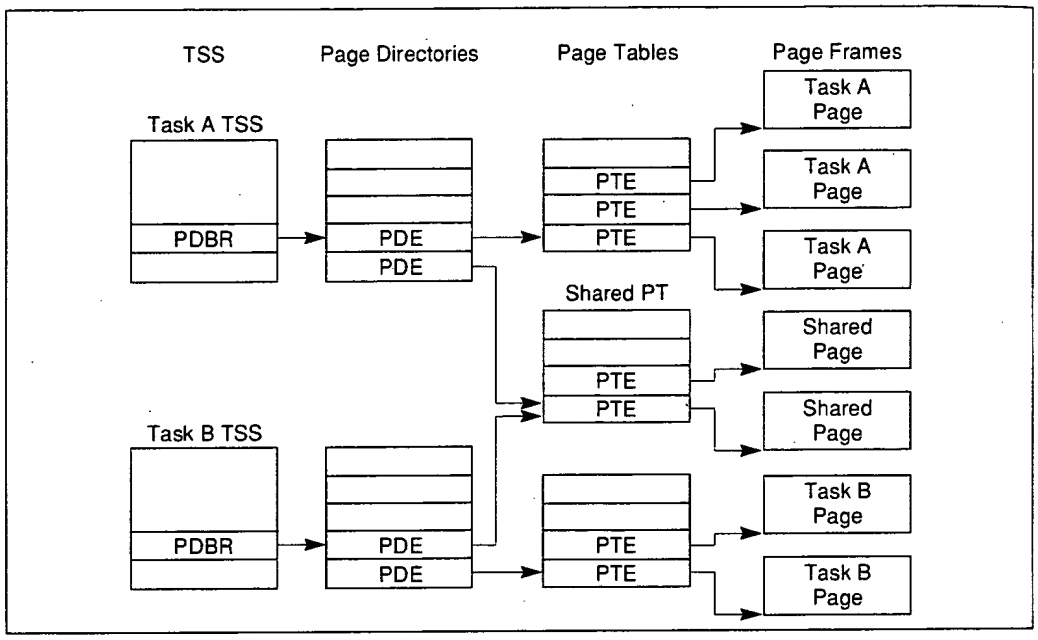


Figure 6-8. Overlapping Linear-to-Physical Mappings

6.5.2. Task Logical Address Space

To allow the sharing of data among tasks, use any of the following techniques to create shared logical-to-physical address-space mappings for data segments:

- Through the segment descriptors in the GDT. All tasks must have access to the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear-address space that are mapped into an area of the physical-address space common to all tasks, then all tasks can share the data and code in those segments.
- Through a shared LDT. Two or more tasks can use the same LDT if the LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments that are mapped to a common area of the physical address space, the data and code in those segments can be shared among the tasks that share the LDT. This method of sharing is more selective than sharing through the GDT, because the sharing can be limited

to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared segments.

- Through segment descriptors in distinct LDTs that are mapped to common addresses in the linear address space. If this common area of the linear address space is mapped to the same area of the physical address space for each task, these segment descriptors permit the tasks to share segments. Such segment descriptors are commonly called aliases. This method of sharing is even more selective than those listed above, because, other segment descriptors in the LDTs may point to independent linear addresses which are not shared.

6.6. 16-BIT TASK-STATE SEGMENT (TSS)

The 32-bit Intel Architecture processors also recognize a 16-bit TSS format like the one used in Intel 286 processors (see Figure 6-9). It is supported for compatibility with software written to run on these earlier Intel Architecture processors.

The following additional information is important to know about the 16-bit TSS.

- Do not use a 16-bit TSS to implement a virtual-8086 task.
- The valid segment limit for a 16-bit TSS is 2CH.
- The 16-bit TSS does not contain a field for the base address of the page directory, which is loaded into control register CR3. Therefore, a separate set of page tables for each task is not supported for 16-bit tasks. If a 16-bit task is dispatched, the page-table structure for the previous task is used.
- The I/O base address is not included in the 16-bit TSS, so none of the functions of the I/O map are supported.
- When task state is saved in a 16-bit TSS, the upper 16 bits of the EFLAGS register and the EIP register are lost.
- When the general-purpose registers are loaded or saved from a 16-bit TSS, the upper 16 bits of the registers are modified and not maintained.

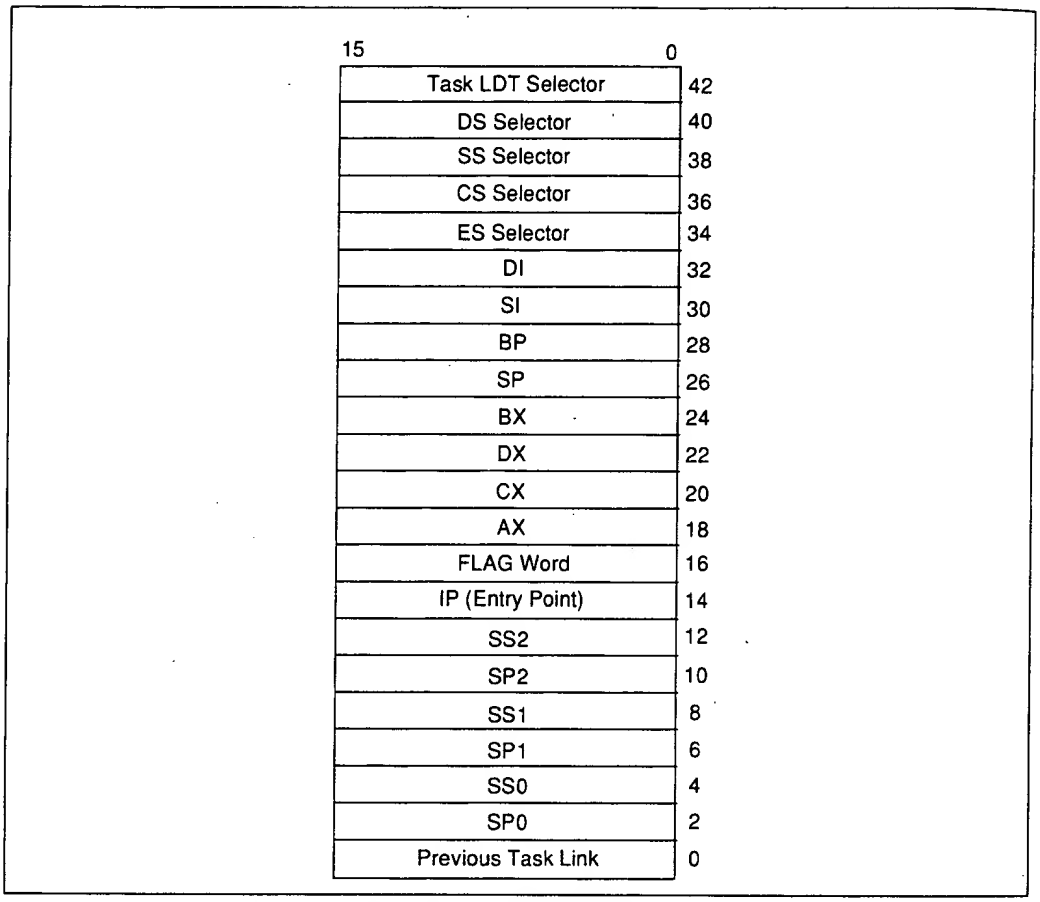


Figure 6-9. 16-Bit TSS Format

Thread (computer science)

From Wikipedia, the free encyclopedia.

Many programming languages, operating systems, and other software development environments support what are called "**threads**" of execution. Threads are similar to processes, in that both represent a single sequence of instructions executed in parallel with other sequences, either by time slicing or multiprocessing. Threads are a way for a program to split itself into two or more simultaneously running tasks. (The name "thread" is by analogy with the way that a number of threads are interwoven to make a piece of fabric).

A common use of threads is having one thread paying attention to the graphical user interface, while others do a long calculation in the background. As a result, the application more readily responds to user's interaction.

An unrelated use of the term **thread** is for threaded code, which is a form of code consisting entirely of subroutine calls, written without the subroutine call instruction, and processed by an interpreter or the CPU. Two threaded code languages are Forth and early B programming languages.

Contents

Threads compared with processes

Threads are distinguished from traditional multi-tasking operating system processes in that processes are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Multiple threads, on the other hand, typically share the state information of a single process, share memory and other resources directly. Context switching between threads in the same process is typically faster than context switching between processes. Systems like Windows NT and OS/2 are said to have "cheap" threads and "expensive" processes, while in other operating systems there is not so big a difference.

An advantage of a multi-threaded program is that it can operate faster on computer systems that have multiple CPUs, or across a cluster of machines. This is because the threads of the program naturally lend themselves for truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require atomic operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.

Use of threads in programming often causes a state inconsistency. A common anti-pattern is to set a global variable, then invoke subprograms that depend on its value. This is known as accumulate and fire.

Operating systems generally implement threads in one of two ways: preemptive multithreading, or cooperative multithreading. Preemptive multithreading is generally considered the superior implementation, as it allows the operating system to determine when a context switch should occur. Cooperative multithreading, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing priority inversion or other bad effects which may be avoided by cooperative multithreading.

Traditional mainstream computing hardware did not have much support for multithreading as switching between threads was generally already quicker than full process context switches. Processors in embedded systems, which have higher requirements for real-time behaviors, might support multithreading by decreasing the thread switch time, perhaps by allocating a dedicated register file for each thread instead of saving/restoring a common register file. In the late 1990s, the idea of executing instructions from multiple threads simultaneously has become known as simultaneous multithreading. This feature was introduced in Intel's Pentium 4 processor, with the name *Hyper-threading*.

Processes, threads, and fibers

The concept of a *process*, *thread*, and *fiber* are interrelated by a sense of "ownership" and of containment.

A *process* is the "heaviest" unit of kernel scheduling. Processes own resources allocated by the operating system. Resources include memory, file handles, sockets, device handles, and windows. Processes do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way. Processes are typically pre-emptively multitasked. However, Windows 3.1 and older versions of Mac OS used co-operative or non-preemptive multitasking.

A *thread* is the "lightest" unit of kernel scheduling. At least one thread exists within each process. If multiple threads can exist within a process, then they share the same memory and file resources. Threads are pre-emptively multitasked if the operating system's process scheduler is pre-emptive. Threads do not own resources except for a stack and a copy of registers including the program counter.

In some situations, there is a distinction between "kernel threads" and "user threads" -- the former are managed and scheduled by the kernel, whereas the latter are managed and scheduled by thread tools and scheduling in userspace. In this article, the term "thread" is used to refer to kernel threads, whereas "fiber" is used to refer to user threads.

A *fiber*, also known as a coroutine, is a user-level thread. Fibers are co-operatively scheduled: a running fiber must explicitly "yield" to allow another fiber to run. A fiber can be scheduled to run in any thread in the same process.

Thread and fiber issues

Typically fibers are implemented entirely in userspace. As a result, context switching between fibers in a process is extremely efficient: because the kernel is oblivious to the existence of fibers, a context switch does not require a system call. Instead, a context switch can be performed by saving the CPU registers used by the currently executing fiber, and loading the registers required by the fiber to be executed. Since scheduling occurs in userspace, specific scheduling mechanisms can be tailored for a certain task, by the userlevel program.

However, the use of blocking system calls (such as are commonly used to implement synchronous I/O) in fibers can be problematic. If a fiber performs a system call that blocks (perhaps to wait for an I/O operation to complete), the other fibers in the process are unable to run until the system call returns.

A common solution to this problem is providing an I/O API that implements a synchronous interface by using non-blocking I/O internally, and scheduling another fiber while the I/O operation is in progress. Win32 supplies a fiber API. SunOS 4.x implemented "light-weight processes" or LWPs as fibers known as "green threads". SunOS 5.x and later, NetBSD 2.x, and DragonFly BSD implement LWPs as threads as well.

Alternatively, a system call such as `select` under Unix and Unix-like operating systems can be used to check whether certain system calls will block, but this adds complexity to the runtime system.

The use of kernel threads brings simplicity; the program doesn't need to know how to manage threads, as the kernel handles all aspects of thread management. There are no blocking issues since if a thread blocks, the kernel can reschedule another thread from within the process or from another, nor are extra system calls needed.

However, there are obvious issues with managing threads through the kernel, since on creation and removal, a context switch between kernel and usermode needs to occur, so programs that rely on using a lot of threads for short periods may suffer performance hits.

Hybrid schemes are available which gains a tradeoff between the two.

Relationships between processes, threads, and fibers

The operating system creates a process for the purpose of running a program. Every process has at least one thread. On some operating systems, processes can have more than one thread. A thread can use fibers to implement cooperative multitasking to divide the thread's CPU time for multiple tasks. Generally, this is not done because threads are cheap, easy, and well implemented in modern operating systems.

Processes are used to run an instance of a program. Some programs like word processors are designed to have only one instance of themselves running at the same time. Sometimes, such programs just open up more windows to accommodate multiple simultaneous use. After all, you can go back and forth between five documents, but you can edit one of them at a given instance.

Other programs like command shells maintain a state that you want to keep separate. Each time you open a command shell in Windows, the operating system creates a process for that shell window. The shell windows do not affect each other. Some operating systems support multiple users being logged in simultaneously. It is typical for dozens or even hundreds of people to be logged into some Unix systems. Other than the sluggishness of the computer, the individual users are (usually) blissfully unaware of each other. If Bob runs a program, the operating system creates a process for it. If Alice then runs the same program, the operating system creates another process to run Alice's instance of that program. So if Bob's instance of the program crashes, Alice's instance does not. In this way, processes protect users from failures being experienced by other users.

However, there are times when a single process, instance of a program, needs to do multiple things asynchronously. The quintessential example is a program with a graphical user interface (GUI). The program must repaint its GUI and respond to user interaction even if it is currently spell-checking a document or playing a song. For situations like these, threads are used.

Threads allow a program to do multiple things concurrently. Since the threads a program spawns share the same address space, one thread can modify data that is used by another thread. This is both a good and a bad thing. It is good because it facilitates easy communication between threads. It can be bad because a poorly written program may cause one thread to inadvertently overwrite another data being used by another thread. The sharing of a single address space between multiple threads is one of the reasons that multithreaded programming is usually considered to be more difficult and error-prone that programming a single-threaded application.

There are other potential problems as well such as deadlocks, livelocks, and race conditions. However, all of these problems are concurrency issues and as such affect multi-process and multi-fiber models as well.

Threads are also used by web servers. When a user visits a web site, a web server will use a thread to serve the page to that user. If another user visits the site while the previous user is still being served, the web server can serve the second visitor by using a different thread. Thus, the second user does not have to wait for the first visitor to be served. This is very important because not all users have the same speed Internet connection. A slow user should not delay all other visitors from downloading a web page. For better performance, threads used by web servers and other Internet services are typically pooled and reused to eliminate even the small overhead associated with creating a thread.

Fibers were popular before threads were implemented by the kernels of operating systems. Historically, fibers can be thought of as a trial run at implementing the functionality of threads. There is little point in using fibers today because threads can do everything that fibers can do and threads are implemented well in modern operating systems.

Implementations

There are many different and incompatible implementations of threading. These can either be kernel-level or user-level implementations. For example, the Linux kernel does not treat threads differently from processes, but the C library implements threading and emulates it using processes. (NB: Linux has CoW fork(), so this is efficient.)

Kernel-level

- LWKT in various BSDs.
- M:N threading ?? (in BSDs)

User-level

- NPTL Native Posix Threading Library for Linux from Red Hat. (What makes it "native"? Buzzword?)
- Pthread ??
- Java threads Java emulates threading for multi-threaded applications
- Python threads Similar to Java's approach (??)

Comparison between models

Multiprocess	Multithreaded	Fibers	Example
No	No	No	A program running on DOS. The program can only do one thing at a time.
No	No	Yes	Windows 3.1 running on top of DOS. Every program is run in a single process, and so programs can corrupt each other's memory space. This happened often and caused the infamous General Protection Fault. A poorly written program could easily crash Windows 3.1 because there was only one process. Early versions of Mac OS also fall into this category.
No	Yes	No	Theoretically possible, but no known examples. This is not a useful case. If an operating system supports threads, it almost definitely support multiple processes.
No	Yes	Yes	Theoretically possible, but no known examples. This is not a useful case.
Yes	No	No	Most early implementations of Unix. The operating system could run more than one program at a time, and executing programs were protected from each other. If a program behaved badly, it could crash its process ending that instance of the program. However, the operating system and other programs would not be disrupted in most cases. However, sharing information between processes became necessary. Unfortunately, doing so in this model is awkward and error-prone involving techniques like shared memory. If a single program needed to perform multiple tasks asynchronously, a copy of the process had to be made using the expensive fork() function.
Yes	No	Yes	Sun OS before Solaris. Sun OS is Sun Microsystem's version of Unix. Sun OS implemented "green threads" in order to allow a single process to asynchronously perform multiple tasks such as playing a sound, repainting a window, and responding to user events such as clicking the stop button. Although processes were pre-emptively scheduled, the "green threads" or fibers were co-operatively multitasked. Often this model was used before real threads were implemented. This model is still used in microcontrollers and embedded devices.
Yes	Yes	No	This is the most common case for applications running on Windows NT, Windows 2000, Windows XP, Mac OS X, Linux, and other modern operating systems. Although each of these operating systems allows the programmer to implement fibers or use a fiber library, most programmers do not use fibers in their applications. The programs are multithreaded and run inside a multitasking operating system, but perform no user-level context switching. On the typical home computer, most running processes have two or more threads. A few processes will have a single thread. Usually these processes are services running without user interaction. Typically there are no processes using fibers.
Yes	Yes	Yes	Pretty much all operating systems after 1995 fall into this category. The use of threads to perform concurrent operations is the most common choice, although there are also multi-process and multi-fiber applications. Threads are used to enable a program to render its graphical user interface while waiting for input from the user or performing a task like spell checking. Note that fibers can be implemented without operating system support, although some operating systems or libraries provide explicit support for them. For example, recent versions of Microsoft Windows support a fiber API for applications that want to gain performance improvements by managing scheduling themselves, instead of relying on the kernel scheduler (which may not be tuned for the application). Microsoft SQL Server 2000's user mode scheduler, running in fibre mode, is an example of doing this. It is also worth noting that many software developers believe that in most cases attempts to use fibers actually decrease the performance of an application. There are several reasons for this. First, the use of fibers does not increase the percentage of CPU time that the operating

			sytem gives to a process. Second, in typical multithreaded or multifibered code there is contention for shared resources like variables. Programming constructs like semaphores and critical sections are used to solve problems that multithreading and multifibering create. Often these tools are implemented more efficiently in the operating system than in user libraries, particularly if those libraries are not written in assembly language. Third, as users install newer versions of an operating system, improvements may be made to the kernel scheduler, operating system provided semaphores, etc. User libraries do not see these benefits unless they call libraries provided by the operating system. When fibers are used, it is typically on a computer configured as a server rather than a client.

Example of multithreaded code

This is an example of a simple multi-threaded program written in Java. The program calculates prime numbers until the user types the word "stop". Then the program prints how many prime numbers it found and exits. This example demonstrates how threads can access the same variable while working asynchronously. This example also demonstrates a simple "race condition". The thread printing prime numbers continues to do so for a short time after the user types "stop". Of course, this problem is easily corrected using standard programming techniques.

```
import java.io.*;

public class Example implements Runnable
{
   static Thread threadCalculate;
   static Thread threadListen;
   long totalPrimesFound = 0;

   public static void main(String[] args)
   {
       Example e = new Example();

       threadCalculate = new Thread(e);
       threadListen = new Thread(e);

       threadCalculate.start();
       threadListen.start();
   }

   public void run()
   {
       Thread currentThread = Thread.currentThread();

       if (currentThread == threadCalculate)
           calculatePrimes();
       else if (currentThread == threadListen)
           listenForStop();
   }

   public void calculatePrimes()
   {
       int n = 1;

       while (true)
       {
           n++;
           boolean isPrime = true;

           for (int i = 2; i < n; i++)
               if ((n / i) * i == n) // (n / i) does return an int, not a float!
               {
                   isPrime = false;
                   break;
               }

           if (isPrime)
           {
               totalPrimesFound++;
               System.out.println(n);
           }
       }
   }
```

```
    public void listenForStop()
    {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        String line = "";

        while (!line.equals("stop"))
        {
            try
            {
                line = input.readLine();
            }
            catch (IOException exception) {}
        }

        System.out.println("Found " + totalPrimesFound +
            " prime numbers before you said stop");
        System.exit(0);
    }
}
```

The spin lock article includes a C program using two threads that communicate through a global integer.

See also

- List of multi-threading libraries
- clone()
- Communicating sequential processes
- completion port
- fork()
- Hyperthreading™
- Lock-free and wait-free algorithms
- Message passing
- priority inversion
- synchronization
- Thread safety
- Threading model
- worker
- SEDA

References

- David R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, ISBN 0-201-63392-2
- Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farell: *Pthreads Programming*, O'Reilly & Associates, ISBN 1-56592-115-1
- Charles J. Northrup: *Programming with UNIX Threads*, John Wiley & Sons, ISBN 0-471-13751-0
- Mark Walmsley: *Multi-Threaded Programming in C++*, Springer, ISBN 1-85233-146-1
- Paul Hyde: *Java Thread Programming*, Sams, ISBN 0-672-31585-8
- Bill Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, ISBN 0-1-3443698-9
- Steve Kleiman, Devang Shah, Bart Smaalders: *Programming With Threads*, SunSoft Press, ISBN 0-13-172389-8
- Pat Villani: *Advanced WIN32 Programming: Files, Threads, and Process Synchronization*, Harpercollins Publishers, ISBN 0-87930-563-0
- Jim Beveridge, Robert Wiener: *Multithreading Applications in Win32*, Addison-Wesley, ISBN 0-201-44234-5
- Thuan Q. Pham, Pankaj K. Garg: *Multithreaded Programming with Windows NT*, Prentice Hall, ISBN 0-131-20643-5
- Len Dorfman, Marc J. Neuberger: *Effective Multithreading in OS/2*, McGraw-Hill Osborne Media, ISBN 0070178410
- Alan Burns, Andy Wellings: *Concurrency in ADA*, Cambridge University Press, ISBN 0-521-62911-X
- Uresh Vahalia: *Unix Internals: the New Frontiers*, Prentice Hall, ISBN 0-13-101908-2

External links

- Real World Tech article by Paul DeMone (*http://www.realworldtech.com/page.cfm?ArticleID=RWT122600000000*) - Explaining different types of multithreading, hardware implementation requirements and the impact on software.
- Ars Technica article about multithreading, etc (*http://arstechnica.com/paedia/h/hyperthreading/hyperthreading-1.html*)
- Page 1 (*http://www.ece.utexas.edu/~valvano/EE345M/view04.pdf*) & Page 2

(*http://www.ece.utexas.edu/~valvano/EE345M/view05.pdf*), a preemptive multithreaded implementation described

- Forum (*news:comp.programming.threads*)
- Answers to frequently asked questions for comp.programming.threads (*http://www.serpentine.com/~bos/threads-faq/*)
- Frequently Asked Questions (*http://lambdacs.com/cpt/FAQ.html*), Most FAQ (*http://lambdacs.com/cpt/MFAQ.html*)
- Discussion "Writing a scalable server" (*http://groups.google.com/groups?group=comp.programming.threads&threadm=580fae16.0312210310.1410bf2b%40posting.google.com*)
- The C10K problem (*http://www.kegel.com/c10k.html*)
- Business logic processing in a socket server - thread pools (*http://www.jetbyte.com/portfolio-showarticle.asp?articleId=38&catId=1&subcatId=2*)
- System support for scalable network servers (*http://www.cs.rice.edu/CS/Systems/ScalaServer/*)
- cohort scheduling (*http://citeseer.nj.nec.com/larus02using.html*)
- Article "Multi-threading basics (*http://www.niccolai.ws/works/articoli/art-multithreading-en-1a.html*)" by Giancarlo Niccolai
- Article "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software (*http://gotw.ca/publications/concurrency-ddj.htm*)" by Herb Sutter
- An Implementation of Scheduler Activations on the NetBSD Operating System (*http://web.mit.edu/nathanw/www/usenix/*)
- DragonFly - Light Weight Kernel Threading Model (*http://www.dragonflybsd.org/goals/threads.cgi*)
- Java(TM) and Solaris(TM) Threading (*http://java.sun.com/docs/hotspot/threads/threads.html*)

Categories: Computer terminology

Deprecated: Mac OS X < 10.3.x

NAME

sigsetjmp, **siglongjmp**, **setjmp**, **longjmp**, **_setjmp**, **_longjmp**, **longjmperror** - non-local jumps

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <setjmp.h>

int
sigsetjmp(sigjmp_buf env, int savemask);

void
siglongjmp(sigjmp_buf env, int val);

int
setjmp(jmp_buf env);

void
longjmp(jmp_buf env, int val);

int
_setjmp(jmp_buf env);

void
_longjmp(jmp_buf env, int val);

void
longjmperror(void);
```

DESCRIPTION

The **sigsetjmp**(), **setjmp**(), and **_setjmp**() functions save their calling environment in *env*. Each of these functions returns 0.

The corresponding **longjmp**() functions restore the environment saved by their most recent respective invocations of the **setjmp**() function. They then return so that program execution continues as if the corresponding invocation of the **setjmp**() call had just returned the value specified by *val*, instead of 0.

Pairs of calls may be intermixed, i.e. both **sigsetjmp**() and **siglongjmp**() and **setjmp**() and **longjmp**() combinations may be used in the same program, however, individual calls may not, e.g. the *env* argument to **setjmp**() may not be passed to **siglongjmp**().

The **longjmp**() routines may not be called after the routine which called the **setjmp**() routines returns.

All accessible objects have values as of the time **longjmp**() routine was called, except that the values of objects of automatic storage invocation duration that do not have the *volatile* type and have been changed between the **setjmp**() invocation and **longjmp**() call are indeterminate.

The **setjmp**()/**longjmp**() pairs save and restore the signal mask while **_setjmp**()/**_longjmp**() pairs save and restore only the register set and the stack. (See **sigprocmask(2)**.)

The **sigsetjmp**()/**siglongjmp**() function pairs save and restore the signal mask if the argument *savemask* is non-zero, otherwise only the register set and the stack are saved.

ERRORS

If the contents of the *env* are corrupted, or correspond to an environment that has already returned, the **longjmp**() routine calls the routine **longjmperror(3)**. If **longjmperror**() returns the program is aborted (see **abort(3)**). The default version of **longjmperror**() prints the message ``longjmp botch'' to standard error and returns. User programs wishing to exit more gracefully should write their own versions of **longjmperror**().

SEE ALSO

sigaction(2), **sigaltstack(2)**, **signal(3)**

STANDARDS

The **setjmp**() and **longjmp**() functions conform to ISO/IEC 9899:1990 (``ISO C89''). The **sigsetjmp**() and **siglongjmp**() functions conform to IEEE Std 1003.1-1988 (``POSIX.1'').

Deprecated: Mac OS X < 10.3.x